Interface

2004
3

Contents




[ 表紙デザイン：㈱プランニング・ロケッツ ]

```
#include <stdio.h>

char bdat1 = 0;
char bdat2 = 0;

void func1(void)
{
    bdat1 = bdat2 = -1;
    printf("bdat1 = %d, bdat2 = %d¥n",
            bdat1, bdat2);
    scanf("%d", &bdat1);
    printf("bdat1 = %d, bdat2 = %d¥n",
            bdat1, bdat2);
}

void func2(void)
{
    char b1;

    bdat1 = bdat2 = -1;
    printf("bdat1 = %d, bdat2 = %d¥n",
            bdat1, bdat2);
    scanf("%d", &b1);
    bdat1 = b1;
    printf("b1 = %d, bdat1 = %d, bdat2 = %d¥n",
            b1, bdat1, bdat2);
}
```



特集

間違いやすいコーディング例からROM化プログラミングまで

# Cプログラミングの基礎知識



特集執筆：中島 信行(Nobuyuki Nakashima)

# Interface

2004 3

Contents



連載「プログラミングの要」「フリーソフトウェア徹底活用講座」は，お休みさせていただきます．

ユビキタスコンピューティングを支える「T-Engine」

# TRONSHOW 2004

北村 俊之

「ユビキタスTRONに出会う」をキーワードに，TRONの最新技術を集めた展示会「TRONSHOW 2004」が，2003年12月11日(木)～13(土)の3日間，東京国際フォーラムで開催された．主催は，T-Engineフォーラムおよび(社)トロン協会で，39社・団体が出展した．展示会場では，TRONの技術を応用したさまざまな体験型展示や，Windows，Linux，Javaとの融合によるソリューションなど，実用段階にきたといわれる組み込み用次世代標準開発プラットホーム「T-Engine」の最新の成果が数多く展示されていた．

従来の組み込み用OSでは，ミドルウェアはCPUアーキテクチャに縛られていたが，「T-Engine」は「T-Kernel」と呼ばれる標準リアルタイムカーネル上で動作するようにハードウェアの規格を共通化している点が特徴である．

また，2003年は，WindowsやLinuxなど，他のプラットホームとの融合という点でも，業界を大いに活気付かせていた．2003年3月には，T-Engineソフトウェアアーキテクチャで定義される「T-Linux」として「MontaVista Linux」を移植して対応させることで，T-Engineフォーラムとモンタビスタソフトウェアとの間で合意がなされた．また，9月にはマイクロソフトがT-Engineフォーラムに幹事会員として入会することが発表された．これにより情報家電向けプラットホーム「Windows CE.NET」をT-Engineプラットホーム上で動作させる環境が整うこととなり，TRONとWindowsの長所を活用した次世代デバイスへの期待が高まっている．

初日のオープニングセッションでは，東京大学の坂村健教授，米モンタビスタソフトウェアのジム・レディ社長，米マイクロソフトの古川享バイスプレジデントの講演も行われた．また，「ユビキタスIDセンターとオートIDセンターは仲良し」と題する，慶応大学の村井純教授と坂村教授との対談も行われ，来場者の注目を集めていた．

## ● Windows，Linux，Java ── 出揃ったプラットホーム

今回の展示会でもっとも注目を集めていたブースの一つが，マイクロソフトであろう．同ブースでは，技術検証の中間成果発表として，T-KernelとWindows CE.NETの共存環境実現の前段階として，T-Engineボード上でのITRONとWindows CE.NETの協調動作のデモが行われていた．デモ機では，バックグランドでT-Kernelが背景を描画している際中に，フロントエンドでWindows Media Player 9を動作させるというものであった(写真1)．



〔写真1〕T-Engine上で動作するWindows CE.NET

T-Kernelのミドルウェア群の一つ，T-Linuxとして稼動し，Linuxアプリケーションを実行可能なプログラミングサービス「MontaVista Linux」を提供するモンタビスタソフトウェアジャパンでは，T-Engineアーキテクチャ対応の「MontaVista Linux」が稼動するボードの展示を行っていた．「MontaVista Linux」は，x86，PowerPC，XScale，StrongARM，MIPS，SH，ARM，Xtensaを含む多くのアーキテクチャをサポートしているとのことであった．

2003年12月10日に，T-Engineフォーラムとサン・マイクロシステムズは，T-Engine上へのJava実行環境の実装が完了したと発表した．これによって，アプリケーション開発環境の統一や開発効率の向上などが期待できることになる．

「JBlend」によってT-EngineとJavaの融合を推進してきたアプリックスのブースでは，参考出品ながら「Motion Watch」のコンセプトモデルが来場者の関心を集めていた(写真2)．同製品は「時計自身が着替える」という新しいコンセプトで，時計の盤面を液晶ディスプレイで構成し，盤面のデザイン自体は，Javaアプリケーションとしてダウンロードすることになる．これによって，ユーザーの好みに合わせた時計に仕上げることができるという．



〔写真2〕アプリックスの「Motion Watch」

## ● その他の注目製品

NECエレクトロニクスでは，T-Engineの応用製品として開発された「T-Cube」の展示を行っていた(写真3)．同製品は，標準T-Engineボードの小型軽量化を図ったものだ．サイズが52×52×45mm，重さが165gという小型ボディに，RGB出力，LAN，USB×2，CFスロット，AUDIO OUT/MIC IN，eTRONチップスロット，RS-232-Cなどのインターフェースを装備している．CPUは，同社の「$V_R$5500(800MIPS)」に周辺チップを統合化した「$V_R$5701」を搭載しており，$V_R$5500搭載の標準T-Engineボードとほぼ同等の性能を実現しているという．また，グラフィック性能もSXGA(1280×1024画素)を実現している．



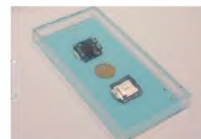
〔写真3〕NECエレクトロニクスの「T-Cube」

BTRON仕様OS「超漢字」関連製品で来場者の高い関心を集めていたのが，パーソナルメディアである．こちらでは，「超漢字4」で動作する「超漢字原稿プロセッサ」が参考出展されていた．「超漢字」の持つ17万字におよぶ文字が使え，推敲，校正作業もサポートし，校正を反映した清書文章のワンタッチ表示などの機能を備えている．展示会初日には，作家の荒俣宏氏が同ブースを訪れ，「超漢字原稿プロセッサ」の機能に熱心に耳を傾けていた(写真4)．



〔写真4〕「超漢字原稿プロセッサ」に興味を示す荒俣宏氏

YRPユビキタス・ネットワーキング研究所では，ユビキタス環境下でネットワークを介してあらゆる機器を制御する末端装置である「nT-Engine」(写真5)のモックアップ展示を行っていた．同装置は，500円玉程度の大きさの基板に各種I/Oを持ったシングルチップマイコンとeTRONチップを搭載している．また，ネットワークプロトコルには機器を制御するための専用プロトコルUNPを使用し，パケットの最大遅延時間を保証しているとのことだった．



〔写真5〕YRPユビキタス・ネットワーキング研究所の「nT-Engine」

展示会場の中央には，ユビキタスコンピューティングを活用した近未来の生活スタイルを紹介する「ユビキタス・ショーケース」(写真6)が設けられていた．こちらでは，eTRONカード(入場券)で，電子マネーによるショッピングや売り場情報の記録などが体験できた．また，T-Engineをベースとした携帯端末「ユビキタスコミュニケータ(UC)」を利用することで，商品や展示物などの情報を音声や画像で確認したり，UCによる家庭内のTVや照明などのリモコン操作などが実際にできるデモも行われていた．



〔写真6〕「ユビキタス・ショーケース」の入り口

## アックス，ユビキタスOSの開発に着手

■ 日時：2003年10月5日（月）
■ 場所：アックス東京支店（東京都千代田区）

（株）アックスがLinuxをベースとしたOS「ユビキタスOS」の開発に着手することを発表し，情報処理振興事業協会（IPA）のオープンソフトウェア活用基盤整備事業として採択された．そこで，同OSについて同社代表取締役の竹岡 尚三氏にインタビューを行った．

ユビキタスOSは，もともとデスクトップOS向けに作られていたLinuxカーネルを，組み込み向けに省電力化を中心に改良する試み．また，カーネル本体の性能を上げることにより，CPUのクロック・スピードを下げられるため，性能向上も間接的に省電力化につながる．



ユビキタスOSのプロジェクトページ http://sourceforge.jp/projects/ubiquitous-os/

手法としては，以下のとおりだ．

● TRAPを使わないシステムコールの実装

Linuxのシステムコールは TRAPによる実装が行われている．これによりアドレス空間の切り替えが発生することなどから，外部バスの駆動が必要となり，電力消費のうえで問題となる．

そこでget_pid()など，一部のシステムコールをマクロで実装し，メモリ参照のみで行えるようにする．参照するメモリはユーザー空間にマップするため，他のタスクにより破壊される心配はない．

● open()/read()/write()をmmap()相当にする

ファイル関係のシステムコールを，ファイルをメモリ空間にマッピングするメモリマップドファイルに置き換える．これによりファイルアクセスが単純なメモリアクセスに置き換わり，なおかつシステムコール自体もマクロで実装するため，性能の向上が期待できる（前述のとおり，これは省電力化につながる）．また，pipe()も同様にマクロ化する．

メモリ上にマップすることにより，ファイルのサイズが論理アドレス空間以下に制限されるが，組み込み用途では実用上問題ないとしている．

● プロトコルスタック，電力制御機構の改良

ネットワークプロトコルスタックにおいて，システムクロックの変動に動的に追従できるようにするほか，電力制御機構を改良し，各種周辺機器の省電力機能を積極的に活用する．

以上の改良を行うことにより，トータルで消費電力30％減を目標としている．同プロジェクトは社外からも幅広く開発者を募集しており，すでに3社が協力しているほか，個人開発者の参加も歓迎するとのことだ．SourceForge.jpで開発が進められ，成果物はオープンソースで公開される．ファーストリリースは2004年2月を予定している．

## リアルタイムDV編集システム「DV Express ST」

■ URL：http://www.min.co.jp/

（株）エム・アンド・アイ ネットワークはPC/ATをベースとしたリアルタイムDV編集システム「DV Express ST」を発売した．AVキャプチャボードとしてCanopus DV Storm3 RTを搭載し，DV出力とともにアナログビデオ出力もリアルタイムで行われる．

ビデオ編集ソフトとしてLet's Edit RT for DVStorm，Canopus EDIUS 1.5を添付し，CPUとハードディスク容量により6種類のラインナップをもつ．同社は従来業務用と家庭用の二極化していたビデオ編集機器市場の中間である，ハイエンドアマチュアや低コストな業務用機器市場を目指すとしている．



リアルタイムDV編集システム「DV Express ST」

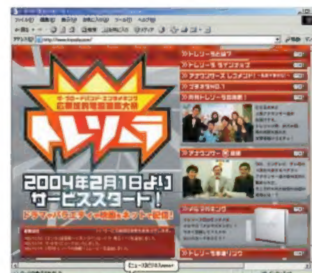
## 民放3局によるブロードバンド映像配信イベント「トレソーラ」

■ 予定日時：2004年2月1日（日）～4月30日（金）
■ URL：http://www.tresola.com/

（株）東京放送（TBS），（株）フジテレビジョン，全国朝日放送（株）（テレビ朝日）の3社により設立された，（株）トレソーラが，2004年2月よりブロードバンド環境を用いたディジタルコンテンツの配信イベントを行う．

同社は2002年にも「Chance!@トレソーラ」と銘打った配信イベントを開催したが，今回の「広帯域的電脳娯楽大帝 ザ・ブロードバンド・エンタメキング・トレソーラ」では，前回のストリーミングによる映像配信に加え，映像データをユーザーにダウンロードさせることによる配信も行うことが特徴だ．これは同社と（株）NTTデータが協力して開発した「トレソーラプレーヤー」によって行う．これはMicrosoft Windows Media 9シリーズの技術を用い，暗号化されたコンテンツファイルをダウンロードしたPCから持ち出せないようにしている．さらにコンテンツファイルの視聴から24時間後，またはダウンロード完了時から7日後には自動消去する機能をもつなど，不正コピー問題などに配慮し，さらにユーザーのストレージを圧迫しないような仕様になっている．

コンテンツは最近放送されたテレビドラマが中心で，月額￥1,000で見放題となっている．




トレソーラのWebページ

移り気な情報工学

# 時代間通信アーキテクチャ

山本 強

早いもので，21世紀もすでに4年目に入った．私は子供のころに21世紀は画期的な科学技術が社会を支えるトンデモない世界になっていると思っていたが，実際に21世紀になってみると，思ったよりも地味なものだと感じている．21世紀になっても自動車は相変わらずタイヤで走っているし，月面にホテルがあるということもない．昔から考えられていたアイテムで実際に実現されたのは携帯電話とインターネット，つまり情報通信ぐらいというのが現実である．

20年前，米国のほんの一握りの研究機関という「点」を結ぶことで始まったインターネットは，今では飛行機の中からでもアクセスできるまでになった．時空間的に見れば，研究機関という点からのアクセス，つまり0次元から始まり，ダイアルアップ接続の1次元，携帯電話で2次元，そして衛星通信で限定的な3次元とアクセス空間の自由度を上げてきたといえる．現時点で物理的にインターネットがアクセスできないのは，地中，海中と宇宙空間ぐらいである．いや，もう一つ残っている軸がある．時間軸である．私たちは21世紀のディジタル情報や文化を100年後，100万年後に正確に送る時間軸の情報伝送法を持っているのだろうか？

## 20年後への情報伝送アーキテクチャ

20年くらい先ならCD-ROMやDVDで情報を保存すれば良さそうに思うが，そう簡単ではない．再生システムや情報フォーマットにも寿命がある．最近でも8インチ，5インチのフロッピディスクやアナログ8mmビデオなどが市場から姿を消しつつある．こういった消えつつある媒体に記録された情報はこの先数年で再生できなくなってしまう．本当に重要なデータはコストをかけてメディアを変換し続けなければならない．タイムカプセルに入れられたCD-ROMやFDはたかだか20年後でも再生を保障するのは難しい．

では，単純にタイムカプセルに入れるだけで20年後にディジタル情報を送るにはどうしたら良いのだろうか．私の知人がこんなことを教えてくれた．きわめて長期間保存しなければならない情報は，PCのハードディスクに書いて，そのPCごと，つまりディスプレイやキーボードまで含めて密閉して倉庫の奥深くしまっておくのだとか．OSは何でも良い．20年後にそのOSやフォーマットが使われなくなっても，記録された情報はディジタルに再生できることになる．簡単なプログラム開発環境も一緒に入れておけば，20年後のフォーマットに変換するプログラムもその環境の上で書けることになる．これはなかなか鋭い話であると感心した．100年ぐらい寿命がある部品でPCそのものを作ることで，この方法は世紀オーダの時代間通信を実現できそうである．

## 1000年後への情報伝送アーキテクチャ

1000年後となると状況はさらに悪化する．現代のエレクトロニクス機器で1000年の寿命を保証できるものは見かけない．1000年後をターゲットにする情報伝送アーキテクチャは半永久的な情報保持機構と受動的な再生メカニズムを持たねばならないのである．でもSFや映画の中にはそういうシステムが描かれている．映画「トゥームレイダー2」ではパンドラの箱の在処を映像で伝える「結晶」が出てくるが，これを再生するときには特別なリーダは必要ではない．ある条件で光を当てると映像が出てくるのである．3次元結晶構造の中にホログラムとして情報を記録することをイメージしているらしい．しかし，ホログラムメモリはいつまでたっても実用化されない，永遠の次世代メモリなのだが，超時代間通信アーキテクチャとしては可能性があるのである．結晶が割れても，情報がそこそこに再生できるというのも時代間通信向きである．

## 100万年後への情報通信アーキテクチャ

100万年後に向けた情報通信となると状況は想像を超えて深刻になる．姿形あるものに記録された情報はいずれ風化し，最後には消えてしまうので，情報の固定化で対応するのは難しい．可能性のある方法の一つは，定期的に自分の複製を作りながら後世に情報を送る方法である．しかし，複製してくれる第三者がいない．偏在するエネルギや物質だけで記録媒体の自己複製を作るメカニズムが必要になってくる．そんなことが可能だろうか．生命体が遺伝子情報を子孫に伝承するメカニズムは限りなくそれに近いように思えるのである．生命のメカニズムは太古の知的生命体が作った時代間通信アーキテクチャであるという仮説も否定はできない．ただし，このアーキテクチャは，愛嬌として突然変異というビット落ちやビット足しというバグがある．

さて，仮にそうだとして．いったい誰が太古の昔にDNA情報を送ったのかという問題が出てくる．生命体が記録メディアなのだから，それは生命体ではないことになる．

生命体ではない知的物体とは何なのか，こうして，今夜も眠れないことになるのである．

やまもと・つよし　北海道大学大学院工学研究科電子情報工学専攻
計算機情報通信工学講座　超集積計算システム工学分野

# ハッカーの常識的見聞録

広畑 由紀夫



今月の常識

## .NET Developer Conference レポート

☆ 12月9日，10日に開催された「.NET Developer Conference」に筆者も参加し，現在のLonghornおよび開発にかかわるセッションに出席した．今回はその中からとくに興味深いものを紹介する．

### ● PDC2003版Longhorn

.NET Developer Conferenceで配布されたLonghornは，米国PDC2003で配布されたバージョンと同一DVD/CDキット（パッケージの一部は日本語）であり，MSDNサイトにて2003年12月中旬現在では未配布のものも含まれます．しかしながら，Longhornのビルドそのものは同じなので，Whidbey，Yukonを使用せず，Longhorn，およびLonghorn SDKでXAMLといった現在のLonghornプレリリース版でみることのできる機能については，2003年12月中旬現在，MSDN会員サイトで配布されているもので実際に触ってみることができるようです．

特に，MSBUILD/XAMLCのようにコマンドラインで行うものはLonghorn SDKのプレビュー版として配布されているため，そちらを使用して，現在のプレリリース版における実際のコードを見ることができます．これらを参考にすることで，今後の開発の方向性などをみることができます．

### ● WinFS（WinFS is not Windows Future Storage）

WinFSは，その構想段階から話題と謎の多い部分でしたが，今回のカンファレンスにおいてLonghornにおけるWinFSがどのようなものであるのか，その詳細が発表されました．今回紹介されたWinFSの機能とは，SQL Serverの検索エンジンをコアに，コンピュータ内に保存されたドキュメントファイルなどの検索を，ファイル名や時刻以外のキーワードを用いて簡単に検索する手段だとのことです．

こうした機能は，シェル機能として拡張されるだけであれば，とりたてて騒ぐことではないのですが，WinFSは従来のコンポーネントとも協調します．そして，ユーザーインターフェースの拡張のみならず，APIとして他のアプリケーションからも操作されうるものであるという点が，開発者の興味を引きます．

さらに，WinFSは既存のNTFS上に構築されるため，既存のNTFSベースで動作するソフトウェアは，ファイルフォーマットの変更がないため，それほど多くの変更を強いられることなく対応できると予測されます．

### ● Avalon

Avalonは，GUIを担当するフレームワークで，直接的に見ることができるのはXAMLです．XAMLは，従来のプログラミング型GUIでは，実装や実装後のデバッグ，さらにそれらを環境に合わせて動的に動作させていたUI部分を，HTMLやXMLのようなマークアップ型で記述します．そして，UIをプログラミングから分離することで，アプリケーションの開発効率の強化や，より使いやすいユーザーインターフェースの設計に専念できるようになっているようです．

もちろん，Avalonは，XAMLだけではなく各種プログラミングインターフェースを含んでいるので，マネージドコードにおいて，Longhorn以後のGUIを作り出すための手法の一つになるのではないかと思います．XAMLについては，Visual Studio .NET上などでの，今後のリソース定義などへの応用も含めて期待しています．

### ● Indigo

Indigoは，Longhornにおいて通信部分を司るフレームワークです．従来，ソケット通信からプロトコルの実装まで多くの部分をC++言語などで書いていた開発者は多いことでしょう．Indigoでは，そうした通信に関わる部分をまとめ，WhidbeyではIndigoと連携することでパケット単位などの低レベル通信ではなく，SOAPなどを使用したオブジェクト単位での通信を簡潔に実装できるようになります．非常に興味深い通信コンポーネント群なので，いずれ改めて解説します．

### ● Longhornに寄せる期待

今までOSのバージョンアップでは，見た目に使いやすいユーザーインターフェースとしてのシェル機能の拡張や，それらのバックグラウンドとなるAPIなどが独立していることが多く，使いやすい機能を利用するために，未公開のAPIなどを参照するといったことがありました．

Longhornでは，こうしたことが改善され，新しいユーザーインターフェース用APIがサンプルコードとともに公開され，開発にかかわる時間の削減や，よりよいユーザーインターフェースの設計が行いやすいよう，多くの面で改善がなされていると思います．

特に，今回は開発環境との密接なつながりが強調され，限定的とはいえ，早く開発に取りかかることができる情報の提供などが行われていることは，とても良いことだと思います．WhidbeyやYukonのプレリリース版の入手が今回できなかった方も，MSDN会員向けにいずれ提供されると思われます．

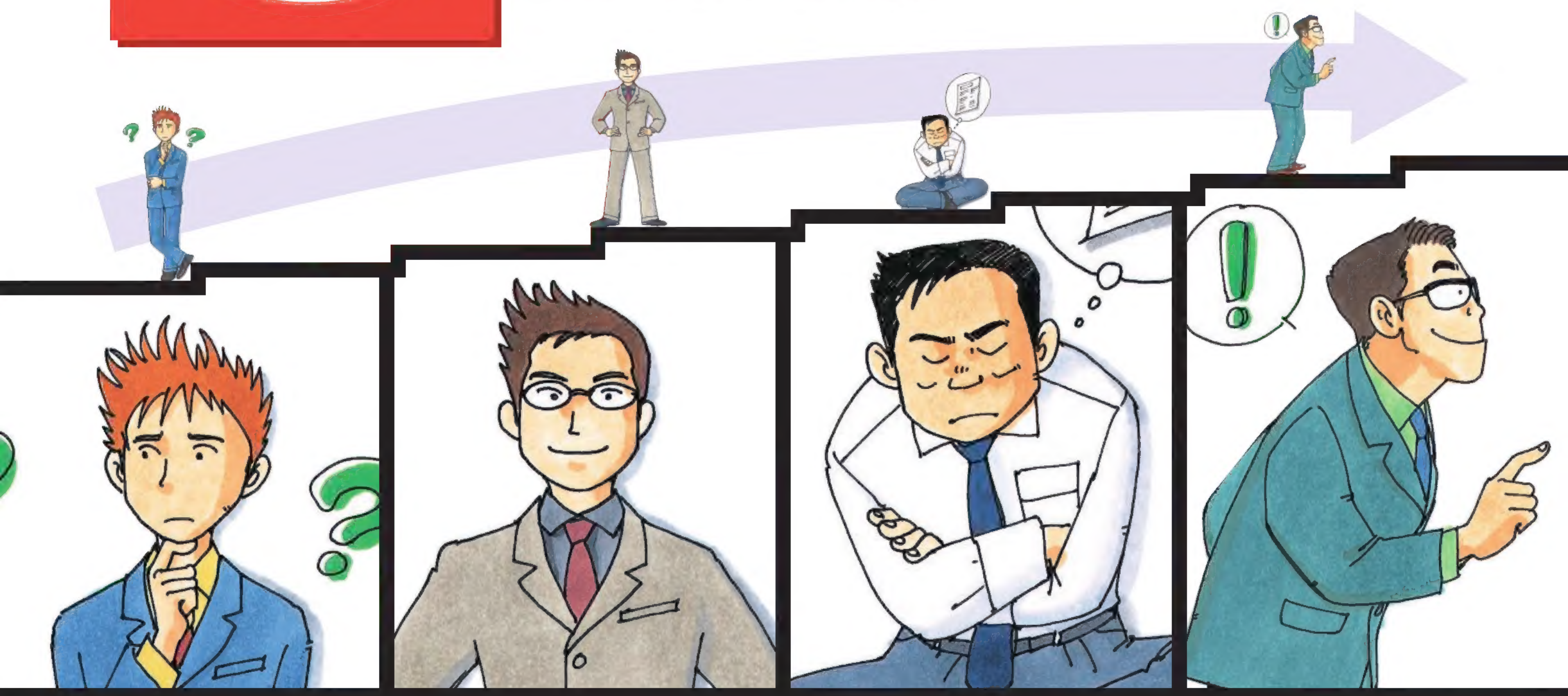
また，2003年12月中旬現在，MSDN会員サイトにおいてLonghornおよびLonghorn SDKの提供がされているため，それらのドキュメントに目をとおしておいたり，日本語情報サイトでチェックしておくと良いでしょう．

ひろはた・ゆきお　OpenLab.

間違いやすいコーディング例からROM化プログラミングまで

# Cプログラミングの基礎知識

特集執筆　中島 信行



C言語の文法をある程度理解した初心者を対象としてCプログラミングに役立つテーマを取り上げて説明します．Cは演算子の記号が特殊で，=と==，&と&&，|と||などのように間違いやすい演算子が少なくありません．これらの演算子は日常的な数学の常識や他の言語と比較してかなり特殊なため，コーディングミスを誘発しやすくなっており，しかも似ている記号が多いためにミスに気づきにくいといった面ももっています．

そこで今回の特集では，まずCで間違いやすいコーディング例を紹介します．さらに，C独特のコーディングのしかたを確認して，処理系の違いがコーディングにどのように影響するか，関数の作成のテクニック，デバッグの方法，ROM化技法などについて解説します．

**プロローグ**

## これからCプログラミングをはじめる人へ

**第1章**　演算子/区切り子/構文の落とし穴/式/関数/引き数/配列/ポインタ/文字列/構造体/マクロ/移植性/浮動小数点

## Cで間違いやすいコーディング例

**第2章**　VC++ 1.5とVC++ 5.0でコンパイル結果を比べてみる

## コーディングの違いと最適化例

**第3章**　ライブラリにしてブラックボックス化する

## 関数作成の勘所

**第4章**　効率よくデバッグする方法を考える

## デバッグの前準備と心得

**第5章**　ROM化するということはどういうことか

## 組み込みCプログラミング

プロローグ

# これから Cプログラミングを はじめる人へ

中島 信行



本特集では，Cの文法をある程度理解した初心者を対象として，Cプログラミングに役立つテーマを取り上げて説明します．

## 筆者がプログラムをはじめたころ

筆者がこの仕事にかかわりはじめたころは8ビット CPUである Z80の全盛期で，簡単なテストプログラムは機械語で打ち込んで動作させることもありました．機械語は CPU が直接理解する言語です．0と 1で表現され，通常は 8進数や 16進数で表示します．この機械語は普通の人間には理解できないため(中には機械語がすべて頭に入っている人もいたが)，実際のプログラムは人間が理解しやすいアセンブリ言語で記述しました．通常はマクロ機能をもったマクロアセンブラで記述して，それをアセンブルして機械語に落とし，ROM 化して，デバッグするといった手順を踏んでいました．

アセンブリ言語(のニーモニック)と機械語は基本的には 1対 1で対応するため，デバッグをしていく過程で勝手に機械語が頭に入ってきて，間違った個所の修正を機械語で行ったりもしたものです．

機械語やアセンブリ言語は CPU に固有のもので，CPU ごとに違うため，68000 や 8086といった高機能な CPU が出てきても，簡単に乗り換えることはできませんでした．

## 組み込みシステム開発での C 言語の普及

筆者がアセンブリ言語でプログラムを組んでいたころでも，C，FORTRAN，Pascal といった高級言語はすでに誕生していて，大型計算機やミニコン，ワークステーションなどで使われていました．また，CP/M と呼ばれるマイコン用の OS で動作する高級言語も存在していましたが，

- 実行速度が遅くなる
- ROM に入らなくなる

といった理由でほとんど使われていませんでした．その後，MS-DOS が全盛のころになると 68000 や 8086といった 16ビット CPU が使われるようになり，ROM の最大容量が 64K バイトから 1M バイトや 16M バイトに増え，CPU の速度も高速になったため，前述の欠点が致命的でなくなり，次第に高級言語が使われるようになりました．

とくに，日本では異常なほど Cコンパイラが流行し C 言語が普及していきました．筆者も Z80のシステムを 68000のシステムに移行した際に Z80のアセンブリ言語のプログラムを 68000の C 言語(一部はアセンブリ言語)のプログラムに書き換えて C 言語を使うようになりました．

一度，C 言語で記述してしまえば，アセンブリ言語よりもプログラミング効率が良いため，その後の大幅な機能追加もかなり楽になりました．このころは C 言語人気から，特殊な CPU や新しく開発された CPU でも，C 言語が利用できる状態になっていました．これが組み込み機器で C 言語が使われる最大の理由です．今では C コンパイラが存在しない CPU を探すほうが困難なほどに，いろいろな CPU に移植されています．

最近，組み込み機器で多く使われるようになった RISC CPU は遅延分岐などのアセンブラプログラミングを難しくする要因があるため，C 言語の使用が前提となっているといった事実もあります．

## C でプログラムを書くメリット

C コンパイラがあるからといっても，C 言語で書かれたプログラムを別の CPU に移植するというのは 68000→68020→68060といった上位の CPU に移植する場合を除けば，一般的にいわれているほど簡単ではありません．CPU の違い以前に，同じ CPU でも，C コンパイラをバージョンアップしたところ，生成コードが変わって動作しなくなったということも多々あります．同じ CPU と同じコンパイラを長い間使っていると，知らないうちにその CPU やそのコンパイラに特有のプログラミングを行っていることが多くなります．とくに，開発者が変わっていくと，初級者から上級者までのいろいろなレベルのプログラムが混ざっていって，移植性が悪くなっていきます．

といっても，どの CPU であっても，慣れた C 言語で記述できるというのは大きなメリットです．

C 言語は，

- 構造化プログラミングに適している
- ポインタで特定のアドレスを直接アクセスできる
- 参考文献が豊富なため，学習しやすい

といった利点もあります．構造化プログラミングは今では古いキーワードとなっていますが，可読性に優れたプログラムを記述できるため，組み込み分野では今でも現役です．

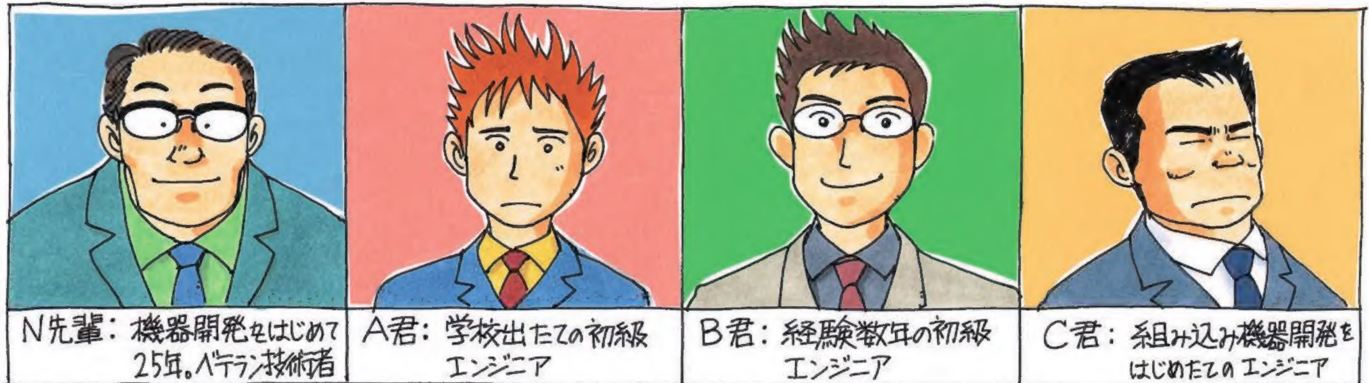
ポインタはバグを生む大きな原因ともなるため，弊害もありますが，組み込み機器では特定のメモリや I/O を直接アクセスしなければならないため，重要な機能です．

## そこで今回の特集の読み方

C 言語の参考文献が豊富という点では，どれを読めばよいのか迷うという弊害もあります．しかし，大きな書店で実際に本を手にとって，自分のレベルに合ったものを探せば，わりと簡単に習得できると思います(インターネットで公開されている文献も少なくない)．

また，C 言語は演算子の記号が特殊で，= と ==，& と &&，| と || などのように間違いやすい演算子が少なくありません．これらの演算子は日常的な数学(算数？)の常識や他の言語と比較してかなり特殊なため，コーディングミスを誘発しやすくなっています．しかも，似ている記号が多いためにミスに気づきにくいといった面もあります．

そこで，第 1 章では主として ANSI C を対象として間違いやすいコーディング例を紹介しています[組み込み分野ではいまだに使用されることがあるカーニハン&リッチー・レベルの C コンパイラ(以後 K&R C と呼ぶ)にも一部で触れている]．これらの間違いやすいパターンを頭においておくことで，ミスを未然に防止するプログラミングを行うことができるようになるはずです．




最近のコンパイラは警告やヒントとして記述ミスに該当しそうな個所をいろいろと指摘してくれるので，コンパイルエラーでないからといって，警告やヒントを無視しないように，できれば警告やヒントがでないように書き換える習慣をつけたほうが良いと思います（N先輩，A君登場）．

第2章ではコーディングの違いと最適化例として，同等の処理をポインタ風コーディングと配列風コーディングとで記述した場合，コンパイラの最適化がどのように違うのかを見ていきます．Cではポインタと配列はコーディング上は同じように記述できる場合があります．ポインタ風のコーディングのほうがCらしいことができるかもしれませんが，場合によっては，読みやすさの点で（とくに，2次元や，3次元などの多次元では）配列風のコーディングが良いこともあります．

コーディングの違いによって，最適化がどのように影響を受けるのか，いくつか例をあげて見ていきます（N先輩，A君登場）．

第3章では関数作成の勘所として，関数の作成にあたっての指針を簡単に紹介していきます．Cのプログラムは関数が基本単位となっており，関数を寄せ集めて，一つのプログラムを構成していきます．役に立つ処理であれば，たとえわずか数行の短いプログラムであっても，独立した関数として記述しておけば，別のプログラムを作成するときにも利用可能となります．そこで，関数の作成にあたって自分なりの指針をもつようにすれば，プログラミングの上達が早くなるものと思います（N先輩，B君登場）．

第4章ではデバッグの前準備の心得として，主としてデバッガを使わない一般的なデバッグ論について紹介していきます．デバッグというのはプログラム作成の最終段階です．しかし，デバッグ段階はまだまだプログラム作成の半ばあたりだと思ったほうが無難です．それだけに，デバッグを行うまでのプロセスや心構えも重要となってきます（N先輩，B君登場）．

第5章では組み込みCプログラミングに関して紹介していきます．MS-DOS/Windows/LinuxなどのOS上のCプログラミングではスタートアップルーチンが標準で組み込まれているため，main関数から書き始めれば，プログラムは動作します．組み込み機器でも，こういったOSを使えば，その組み込み機器上でパソコンと同じようにソースをコンパイルして，実行し，デバッグしていくことが可能となります（N先輩，C君登場）．

組み込みCといっても，Cプログラミングですから，第1章～第4章で紹介する例は組み込みCプログラミングの場合でも同じことです．しかし，組み込みCの対象となる組み込み機器ではキーボードや画面表示をもたないことがあり，その組み込み機器上でソースをコンパイルして実行するということができないため，パソコンやワークステーションなどでプログラムを作成し，コンパイルして実行ファイルを作成します．そして，それを組み込み機器上で実行しデバッグするといったクロス環境での開発が一般的です．

OS上のプログラミングとクロス開発の最初の相違点はmain関数から書き始めたプログラムが組み込み機器で実行できないということです．

じつはCプログラムのmain関数が実行されるまでにはメモリを初期化したり，割り込みベクタを設定したり，…といったスタートアップルーチンと呼ばれる影の下請けプログラムが実行されていて，そこから，main関数が呼び出されているのです．組み込み機器では，パラレル入出力や通信チップなどのハードウェアの初期化なども必要となり，その機器のハードウェアに合わせたスタートアップルーチンが必要となります．

これが，OS上のプログラミングとの最初の相違点となります．初心者の間はスタートアップルーチンができている状態から始めることが多いため，何度か組み込み機器のプログラミングの経験がある人でも，スタートアップルーチンの存在を知らない人がいるかもしれません．しかし，小さな組み込み機器を一人で開発したり，大きな組み込み機器を取りまとめるようになるまでには，スタートアップルーチンを自力で書けるようになっておかなくてはいけません．そのスタートアップルーチンを記述するためには，たとえソフトウェア技術者であっても，ハードウェアの知識が必要となります．

組み込み機器の場合は，ハードウェアを自作することのほうが多いため，ソフトウェアのデバッグ時に渡されるハードウェアは完全に動作しないことが多いため，自分のプログラムをデバッグする前に，ハードウェアの試験プログラムを作成して，ハードウェアの不具合個所をハードウェア設計者に指摘できる程度にはハードウェアを理解できるようにならないと一人前と呼べません．

組み込み機器のプログラマを目指す方はわからなくても，自分のかかわっている組み込み機器のハードウェアの設計図を見せてもらって（できれば簡単にでも説明してもらって），ハードウェアに対する抵抗を徐々になくしていくようにしないといけません（筆者は就職したてのころ，「トランジスタ技術」の回路図がわからなかったが，毎月，眺めているうちにハードウェアに対する抵抗が少なくなり，回路図がそれなりに読めるようになっていった）．

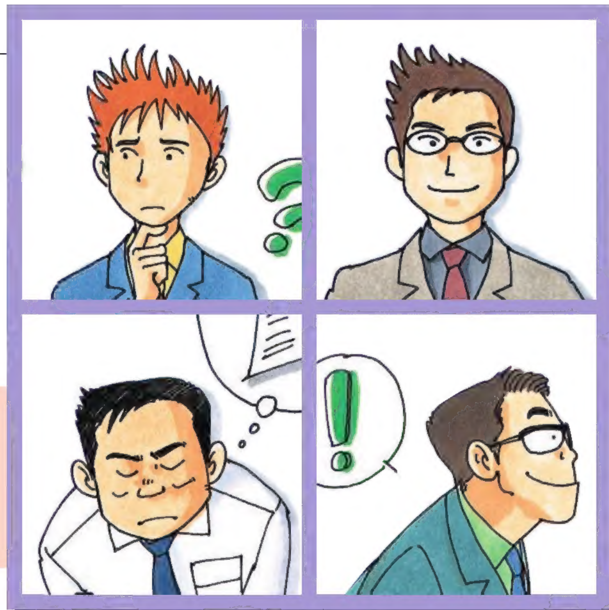
なかしま・のぶゆき　（株）Unix

第1章

演算子/区切り子/構文の落とし穴/式/関数/引き数/配列/ポインタ/文字列/構造体/マクロ/移植性/浮動小数点

# Cで間違いやすいコーディング例

中島 信行



Cはコーディングの自由度が大きく，特殊な演算子も多いため，他の言語とくらべて間違ったコーディングをしてしまうことが少なくありません．本章ではCの初心者A君にN先輩が手ほどきする形でCで間違いやすいコーディング例のいくつかを紹介していきます．

（筆者）

## 演算子編

―― =と ==，&&と&，>>と >，演算子の優先順位

### ● =と == ―― 等価と代入

A君 ：先輩，おはようございます．

N先輩：おはよう．どう，Cプログラミングは少しは慣れたかい？

A君 ：はい．少しは慣れましたが，ミスが多くて…．どこが間違っているのか，わけがわからなくなることも多くあります．

N先輩：そうか．じゃあ間違いやすいコーディング例をいくつか示してあげよう．今後のプログラミングの際に役立ててよ．

A君 ：ぜひ，お願いします．

▶ コンパイラからの警告はないが…

N先輩：まずはリスト1のプログラムを見てごらん．

▶ 等価のつもりが代入になっていた…

N先輩：VC++ 5.0（Visual C++ 5.0）で

```
cl /c /Ot /W3 test.c
```

のようにコンパイルしても“warning”も“error”も出ないプログラムなんだけど，たぶん間違ってコーディングしているんだ．

A君 ：これなら僕もやったことがありますよ．if文の中の=が==の間違いなんですよね．

〔リスト1〕=と ==

```
#include <stddef.h>

char *pointer;
int  flag;

void func(void)
{
    if (pointer = NULL) /* ←もしpointerがNULLであれば */
        flag = 1;       /* ←flagを1にする */
}
```

N先輩：そうだね．僕もやったことがあるけど，=と==のコーディングミスはたいていの人が経験していると思うよ．実はこのミスを予防するコーディング方式があるんだけど，わかるかい．

▶ =と==のミスを避ける方法？

A君 ：うーむ…ちょっとわからないです．

N先輩：実はね，この=と==のミスを避けるために「定数との比較では定数を左側に書く」といった対策をとっている人も少なくないんだよ．たとえばね，

```
if (NULL = pointer)
    /* ... */;
```

のように間違えると代入文の左に定数がくることはないから，コンパイラが指摘してくれるんだ．

A君 ：そう言われてみればそうですね．うまい方法ですけど，コーディングを見ると，ちょっと，変な感じがしますね．

N先輩：確かに，このようなスタイルはプログラムの読みやすさを若干損ねるけど，プロは確実に動くことのほうを重視するものなんだよ．といっても，僕はこのようなコーディングをしないんだけどね（プロと呼べないかもしれませんね）．

A君 ：プロではないからですか？

▶ いまのコンパイラでは“warning”が出るが…

N先輩：えっ，そんな言い方をするかね？ まぁ，いいや．一つ目の理由は，A君が感じたようにちょっと変な感じがするっていう点なんだ．プログラムを読み返すときに違和感があって，そっちのほうに神経が向く可能性があって，肝心のところを見逃すってこともなきにしもあらずだからね．二つ目の理由は，両方変数のときには適用できないってことかな．それで，三つ目のもっとも正当な理由は，最近のコンパイラでは“warning”を出してくれるからだよ．たとえば，C++ Builder 6だったら，

警告 W8060 test.c 8: おそらく不正な代入
( 関数 func)

という"warning"が出る．VC++ 5.0でも"warning"レベルを/W4に上げると，

test.c(8) : warning C4706:
条件式の比較値は，代入の結果になっています．

という"warning"が出るんだ．だから，"warning"を無視しない習慣を付けておけば予防できるミスなんだよ．

A君　：でも，

```
if (*p = *q)
    /* ... */;
```

というようなコーディングをしても"warning"が出てめんどうだと思ったことがありますけど．

N先輩：確かに，"warning"はよけいなお節介と感じる場合も少なくないけど，この"warning"を消すためには，

```
if ((*p = *q) != '¥0')
    /* ... */;
```

といったコーディングをすればいいんだ．"warning"も数が多いと"warning"に対する感覚が麻痺してくるから，できるだけ潰しておいたほうがあとあとケアレスミスで悩むことも少なくなると思うよ．

A君　：はい．そうするようにします．

## ● &&と&　――論理演算子とビット演算子

N先輩：A君は&&と&の違いはわかってるかい?

A君　：&&は論理演算子で真("0"以外)，偽("0")を対象にした論理的なANDで，&はビットごとのANDを取る演算子ですよね．だから，

0x02 && 0x08

は両方の値とも論理値としたら"0"以外だから，結果は真つまり"1"となり，

0x02 & 0x08

はビットごとのANDだから"0"となるんですよね．

N先輩：そういうことだね．それじゃ，リスト2の①と②はどんな違いがあるか説明できるかい．

A君　：①はfunc1()とfunc2()の論理的なANDを取って，flag1は結果として"0"または"1"の値を取りますけど，②はfunc1()とfunc2()のビットごとのANDを取りますよね．

N先輩：だいたい合ってるけど．うーむ，それじゃね，func1()とfunc2()がともに"0"か"1"の値しか返さないときにどういう違いがあるかわかるかな．

A君　：func1()，func2()がともに"0"か"1"の値しか返さないんだったら，両方が"1"を返したときだけflag1もflag2も"1"がセットされるから，とくに違いはないんじゃないですか．

N先輩：それがね，一つだけ大きな違いがあるんだ．

A君　：それは何ですか．

〔リスト2〕&&と&

```
int func1(void);
int func2(void);

int flag1, flag2;

void func(void)
{
    flag1 = func1() && func2(); /* ① 論理AND   */
    flag2 = func1() &  func2(); /* ② ビットAND */
}
```

〔リスト3〕>>と>

```
unsigned udat1, udat2;

void func(void)
{
    udat1 = udat2 > 1;
}
```

▶ &演算子ではつねに後述の関数を評価する

N先輩：リスト2の例でいえば

func1() && func2()

はfunc1()が偽("0")を返したときはfunc2()を実行しないけど，

func1() & func2()

はfunc1()とfunc2()がかならず実行されるという点が大きく違うんだ．

A君　：へえー，そうなんですか．

N先輩：だから，両方の関数を実行させたいときにはわざと②のようなコーディングをすることもあるんだけど，ちょっとわかりにくくなるから，できれば

```
ftmp1 = func1();
ftmp2 = func2();
flag2 = ftmp1 && ftmp2;
```

といったコーディングをしたほうが良いと思うけれどね．これだと二つの関数が実行されることは，明確にわかるよね．

A君　：なるほどね，わかりました．この関係は，||(論理和結合演算子)と|(ビット和演算子)の場合でも同じことですよね．

N先輩：そうだよ．

func1() || func2()

の場合はfunc1()が真(非"0")を返したときにはfunc2()を実行しないけど，

func1() | func2()

はfunc1()とfunc2()がかならず実行されるんだよ．

## ● >>と>　――右シフトと不等号

N先輩：それじゃ，リスト3も簡単なコーディングミスの例だけどわかるかな．

A君　：プログラムを見るかぎりでは，右不等号の>ではなくて，右シフトの>>にすべきではないかと思います．

N先輩：そうだね．>>と>のコーディングミスは1ビット右

シフトの，

    udat >> 1

を，

    udat > 1

と書いてもコンパイルエラーにならないし，けっこう気づきにくいミスなんだ．これはコンパイルスイッチの“warning”レベルを最高にしても“warning”が出ないから，自分で気をつけるしかないと思うよ．

A君　：はい，そうします．

### ● 演算子の優先順位

N先輩：リスト4はBCD 2バイトを10進文字列に変換するプログラムだけど，間違いがあるんだ．どこが間違ってるかわかるかい．

A君　：ふーむ…，最下位4ビットASCII変換のところの，

    val & 0x0f + '0'

があやしそうですね．

N先輩：演算子の優先順位によるトラブルは中級ユーザーでも経験するものだけど．まあ，演算子の優先順位が100%頭の中にはいっている人は少ないだろうからね．僕が経験したものではリスト4の，

    *str = (char)(val & 0x0f + '0');

や，

    val1 = val2 << 4 + val3;

などがあるけど．＋，－の優先順位は意外に高くて，これらは，

    *str = (char)(val & (0x0f + '0'));
    val1 = val2 << (4 + val3);

と解釈されるんだ．だから，リスト4では，

    val & 0x0f + '0'

は，

    (val & 0x0f) + '0'

のように()でくくらないといけないんだ（表1）．

A君　：そうですね．

N先輩：演算子の優先順位はポインタが絡んでくると一層ややこしくなって，

    *ptr[n]

だと[]の優先順位が高いため，

    *(ptr[n]) または *(*(ptr + n))

となるんだ．ポインタそのものが読みにくい性格をもっているから，二重三重のポインタではリストを見直しただけでは気づかないこともあるけどね．優先順位があやふやな場合はマニュアルやオンラインヘルプで確かめることも必要だけど，()でくくるっていう習慣も付けておいたほうが良いだろう．もっとも，()が多くなりすぎると式が見にくくなるから，一時変数を使用して式を分けるというのも対策の一つかもしれないね．最近はコンパイラの最適化の性能が上がってきてるから，式を分けても生成コードが悪くなることは少なくなってきたからね．

A君　：はい，わかりました．

〔表1〕演算子の優先順位

| 演算子 | 結合則 |
|---|---|
| () [] -> . | 左から右へ |
| ! ~ + - ++ -- (型) * & sizeof | 右から左へ |
| * / % | 左から右へ |
| + - | 左から右へ |
| << >> | 左から右へ |
| < <= > >= | 左から右へ |
| == != | 左から右へ |
| & | 左から右へ |
| ^ | 左から右へ |
| | | 左から右へ |
| && | 左から右へ |
| || | 左から右へ |
| ?: | 右から左へ |
| = *= /= %= += -= <<= >>= &= ^= |= | 右から左へ |
| , | 左から右へ |

〔リスト4〕演算子の優先順位（間違い）

```
/*
    BCD 2 バイト → 10 進文字列 (右詰め & '¥0' ターミネート)
*/
void bcdtona(unsigned val,        /* 符号無整数                          */
             char *str,           /* 大 → 小 10進文字列格納アドレスサイズ >= n+1 */
             int  n)              /* 桁数                                */
{
    str += n;                     /* ストリング最後のアドレス +1 セット        */
    *str-- = '¥0';
    do {
        *str-- = (char)(val & 0x0f + '0'); /* 最下位 4ビット Ascii 変換 */
        val >>= 4;
    } while (--n > 0 && val > 0);
    while (--n >= 0)
        *str-- = ' ';             /* スペース セット                      */
}
```

## 区切り子(セパレータ)編
――カンマ(,)の打ち忘れ，余分なセミコロン(;)，足りないセミコロン(;)

### ● カンマ(,)のうち忘れ

N先輩：リスト5は完全なバグなんだけどどこが悪いかわかる

〔リスト5〕カンマ(,)のうち忘れ

```
#include <stdio.h>

void func(void)
{
    static char *msg[] = {
        "abc",
        "def"
        ""
    };
    char **pmsg = msg;

    while (**pmsg)
        printf(*pmsg++);
}
```

かい．

▶ K&R Cではエラーになるのだが…

A君　：うーむ…あっ，わかりました．“def”の後にカンマ（,）がないんですね．

N先輩：古い C（K&R C）だとリスト 5はコンパイルエラーになるんだけど，ANSI Cだと，

```
char str[] = "abc" "def";
```

のように文字列を分けて書いても連結されて，

```
char str[] = "abcdef";
```

と解釈されるから，カンマ（,）を忘れても正常にコンパイルされてしまうんだ．といっても，この仕様は欠点じゃなくて，この仕様のおかげで，

```
#define ESCSTR "¥033"
```

と定義して，

```
ESCSTR "[33m"
```

といった記述ができるようになったから，利点なんだけどね．だけど，リスト 5のように""で文字列定義の最後を判定しようとしてカンマ（,）を付け忘れた場合，コンパイルエラーが出ずに，最後の""が“def”と連結して消えてしまうから，文字列定義の最後が判定できなくなって誤動作してしまうんだ．

▶ 複数の文字列定義の終わりは NULLとする習慣を…

A君　：このミスもなんか良くやりそうな感じですね．何か対策はないですか．

N先輩：完全な対策はないんだけど，こんなミスを犯す場合はたいてい後からターミネータ""の手前に文字列を追加してカンマ（,）を付け忘れるってパターンが多いから，文字列定義の最後を""じゃなくて，NULLにするように変更すれば防げるんだよ．具体的にはリスト 6のようにしておけばカンマ（,）を付け忘れるとコンパイルエラーが発生するから，ミスに簡単に気づくんだけど．もっとも，“abc”の後のカンマ（,）を付け忘れたときなんかのようにコンパイルエラーにならない場合は実行してみて気づくってことになるだろうけどね．

A君　：そうなったら，カンマ（,）の打ち忘れに気づくのに時間がかかることもあるかもしれないですね．

N先輩：この種のエラーを経験していなかったら，時間がかかるだろうね．

A君　：このエラーも頭に入れておきます．

● 余分なセミコロン（;）

N先輩：リスト 7は文字列の最初のスペースをスキップする例だけど，どこがまずいかわかるかい．

A君　：うーむ…あっ，whileの行の右端にセミコロン（;）があるのがまずいんですね．字下げしてあるからわかりにくいですけど，実際は，

```
while (isspace(*p))
    ;
    p++;
```

ってことになりますよね．

〔リスト 6〕カンマ（,）のうち忘れ（コンパイルエラー）

```
#include <stdio.h>

void func(void)
{
    static char *msg[] = {
        "abc",
        "def"
        NULL
    };
    char **pmsg = msg;

    while (*pmsg)
        printf(*pmsg++);
}
```

〔リスト 7〕余分なセミコロン（;）

```
#include <ctype.h>

void func(char *p)
{
    while (isspace(*p));
        p++;
    /* ... */
```

〔リスト 8〕足りないセミコロン（;）

```
int flag;
int data1, data2;

int func(void)
{
    /* ... */
    if (flag)
        return   /* ←ない！ でもエラーにはならない！！ */
    data1 = data2;
    return 0;
}
```

N先輩：よく知られているわりには誰でも一度や二度くらいは経験があるのが余分なセミコロン（;）によるトラブルだよ．リスト 7のように字下げしてあるとなかなか気づかないもんだしね．僕はね，do〜whileではwhileの最後にセミコロン（;）がいるけど，do〜whileをwhile〜に書き換えたときに，このようなミスを犯したことがあるよ．

● 足りないセミコロン（;）

N先輩：セミコロン（;）が足りないときにも問題になることがあるんだ．

A君　：セミコロン（;）を付け忘れたらコンパイルエラーにならないですか．

N先輩：たいていの場合はコンパイルエラーになるんだけど，リスト 8のようにたまたま文法的にあってしまうとなかなか気づかないもんだよ．

リスト 8は，

```
if (flag)
    return data1 = data2;
return 0;
```

となるから“warning”も出ないはずだよ．

A君　：へえー，こんな間違い方もあるなんて怖いですね．

N先輩：Cは自由度が大きいから，こんなパターンはいろいろと考えられると思うよ．

## 構文の落し穴編

―― if～else～endifのネスト，continue文，forループ中のswitch文，defaultのスペルミス

### ● if～else～endifのネスト

N先輩：今度はリスト9だけど，どんな問題が潜んでいるかわかるかい．

A君　：うーむ…ちょっとわかんないですね．

N先輩：それじゃ，リスト9とリスト10はどう違うか説明できるかい．

A君　：えっ～，同じことじゃないんですか．

N先輩：字下げしてあるから，わかりにくいけど，リスト9は，

```
if (flaga) {
    /* ... */;
    if (flagb)
        /* ... */;
    else
        /* ... */;
}
```

となるんだよ．elseは2番目のifに対応することになるんだ．if文の中のif文というのは，最初から書けばこんなふうに間違えることはないだろうけど，後から内側にif文を追加したときに，このミスを犯す場合が多いんだよ．

A君　：これは，よくやりそうですね．

N先輩：これは記述スタイルの問題だから，対策はあるんだよ．たとえば，リスト10のようにif文はかならず{}を付けてブロックにまとめるようにしておくと，このトラブルを未然に防ぐことができるんだよ．

A君　：今度から，そういう習慣を付けるようにします．

N先輩：といっても，僕は{}を付ける習慣はないんだけどね．

A君　：どうしてです？

N先輩：とくに理由はないんだけど，怠惰なんだろうね．僕はキーボードを打つのが速くないから，{}を打つ手間を嫌ってるのかもしれないね．

A君　：なんか他人ごとみたいですね．

N先輩：プログラムを打ち込むときはアルゴリズムの方に神経がいってるから，指のほうは意識してないんだよ．勝手に動いてるって感じかな．だから，他人ごとみたいな言い方になっちゃったけど．

〔リスト9〕if～else～endifのネスト（その1）

```
int flaga, flagb;

void func(void)
{
    if (flaga)
        /* ... */;
        if (flagb)
            /* ... */;
    else
        /* ... */;
}
```

〔リスト10〕if～else～endifのネスト（その2: リスト9との違いは？）

```
int flaga, flagb;

void func(void)
{
    if (flaga) {
        /* ... */;
        if (flagb) {
            /* ... */;
        }
    } else {
        /* ... */;
    }
}
```

〔リスト11〕while内のcontinue文をgoto文に書き換える

```
    while (...) {
        /* ... */;
        if (flag)
            continue;
        /* ... */;
    }
```

（a）問題

```
    while (...) {
        /* ... */;
        if (flag)
            goto label;
        /* ... */;
        label:;
    }
```

（b）解答（continue文→goto文）

〔リスト12〕do～while内のcontinue文をgoto文に書き換える

```
    do {
        /* ... */;
        if (flag)
            continue;
        /* ... */;
    } while (...);
```

（a）問題

```
    do {
        /* ... */;
        if (flag)
            goto label;
        /* ... */;
        label:;
    } while (...) ;
```

（b）解答（continue文→goto文）

〔リスト13〕for内のcontinue文をgoto文に書き換える

```
    for (..; ..;) {
        /* ... */;
        if (flag)
            continue;
        /* ... */;
    }
```

（a）問題

```
    for (..; ..;) {
        /* ... */;
        if (flag)
            goto label;
        /* ... */;
        label:;
    }
```

（b）解答(continue文→goto文)

### ● continue文

N先輩：制御をループの継続部分に移すために使用するcontinue文は知ってるよね．

A君　：はい．

N先輩：だったら，リスト11（a）～リスト13（a）のcontinue文をgoto文で書き換えるとどうなるかやってみてごらん．

A君　：はい…リスト11（b）～リスト13（b）でいいんじゃないですか．

N先輩：立派，立派！　僕はCを始めてしばらくの間はcontinue文がループの先頭に制御を移すというように間違って覚えていたから，do～whileだとwhileの条件式じゃなくて，doの箇所にジャンプするものと思い込んでいたんだよ．

A君　：へえー，先輩でもそんな間違いをすることがあるんですね．

N先輩：誰でも，最初は初心者なんだよ．

A君　：そりゃそうですね．

### ● forループ中のswitch文

N先輩：forループ中にswitch文があるとbreak文と

〔リスト 14〕forループ中のswitch文

```
int stat, flag;

void func(void)
{
    int n;

    for (n = 0; n < 100; n++) {
        /* ... */;
        switch (stat) {
        case 'A':
            /* ... */;
            if (flag)
                break;      /* ① */
            /* ... */;
            break;
        case 'B':
            /* ... */;
            if (flag)
                continue;   /* ② */
            /* ... */;
            break;
        /* ... */;
        }
        /* ... */;
    }
}
```

〔リスト 15〕defaultのスペルミス

```
int databits;

void func(void)
{
    switch (databits) {
    case 8 :
        /* ... */
        break;
    case 7 :
        /* ... */
        break;
    case 6 :
        /* ... */
        break;
    case 5 :
        /* ... */
        break;
    defalut :
        /* ... */
        break;
    }
}
```

continue文の見通しが悪くなるんだけど，リスト14の①のbreak文はforに対するbreakか，switchに対するbreakかわかるかな．

A君　：えっと…switch文に対するbreakですよね．

N先輩：ピンポン．それじゃ②のcontinue文は説明できるかい．

A君　：えっと…continue文はswitch文と関係ないから，forループに対するcontinueですよね．

N先輩：そういうことだね．リスト14だと短いからわかるけど，switch文の中身が長くなってくると，このことを見落としてしまうことがあるから注意しないといけないよ．リスト14のような場合や多重ループの内側からループ外に抜け出すときにはgoto文を使用したほうがスッキリする場合もあるんだ(コラム)．

A君　：はい，わかりました．

● defaultのスペルミス

N先輩：今度も非常に単純な問題だよ．リスト15の間違いがわかるかい．

A君　：うーむ…

N先輩：スペルに注意すればわかるんだけど．

## goto文について　Column

最近はFORTRANやBASICなどでも構造化機能がサポートされているため，たいていの言語ではgoto文を使用する必要がなくなってきています．Cでgoto文の親戚を上げてみると，

(1) goto文
(2) break文
(3) continue文
(4) 関数途中の条件判断によるreturn文

などがあります．(4)はジャンプ先が明確なため，見通しが悪くなることはありませんが，(1)～(3)は乱用するとプログラムが読みにくくなります〔ただし，(4)は後から，最後に(returnの直前に何らかの処理を)追加する必要が生じたときにreturnの一部を見落とす可能性があるという欠点がある〕．

多重ループの内側からループ外に抜け出すときにgoto文の使用を避けてもbreak文が羅列するようだとかえってプログラムが読みにくくなります．多重ループからループ外に抜け出す必要が生じたときは，その処理を関数として切り分けてreturnで逃げるようにするとプログラムも多少，読みやすくなることがあります．

▶ 予約語がラベルになるとき…

A君　：あっ!!　defaultのスペルが間違ってるんですね．

N先輩：defaultのスペルを間違えてもラベルになるだけで文法的にはあってるから，すんなりとコンパイルできちゃうんだ．もっとも，“warning”レベルを上げると定義したラベルが参照されていないというメッセージを表示してくれるコンパイラも少なくないけどね．

A君　：“warning”レベルは，できるだけ最高にしておいたほうが安全なんですね．

N先輩：“warning”レベルを最高にすると煩わしいことも多いけど“warning”レベルを最高にしてコンパイルすることを社内規定にしているところもあるみたいだよ．

## 式編

――式の評価順，小数点(.)とカンマ(,)，整数演算のオーバフロー

● 式の評価順

N先輩：リスト16には誤動作する要因があるんだけど，どこが問題かわかるかい．

▶ 式の評価順がコンパイラによって異なる？

A君　：うーむ…どこが違ってるんですか．

N先輩：それじゃ，ちょっと質問の方法を変えてみよう．リスト16の二つの関数subfuncはどちらが先に実行されるか，わかるかな．

A君　：えーっと…左の方だと思いますけど．

N先輩：Cでは演算子の優先順位や作用の仕方は規定されているんだけど，式の評価順に規定のないものがあるんだ．というよりも評価順序が規定されているのは，

&&
||
?:
,

の四つだけなんだ．&&と||は説明したけど，左のオペランドを最初に評価して，その結果によって右のオペランドを評価するんだ．?:演算子は，

a ? b : c

のように三つのオペランドをとって，最初にaを評価して，その結果に応じてbまたはcを評価するんだ．カンマ演算子は左のオペランドを評価してその値を捨てて，右のオペランドを評価するんだ（表2）．ちょっと，話がそれるけど，

func1(a, b)

と，

func2((a, b))

の違いがわかるかい．

A君　：えーっと…

N先輩：func1(a, b)の，は引き数を区切ってるだけで，カンマ演算子じゃないんだけど，func2((a, b))の，はカンマ演算子になるんだ．

〔リスト16〕式の評価順（間違い）

```
unsigned subfunc(void);

void func(void)
{
    unsigned long lval;

    lval = (unsigned long)subfunc() * 0x10000L + subfunc();
    /* ... */;
}
```

〔リスト17〕式の評価順（正解）

```
unsigned subfunc(void);

void func(void)
{
    unsigned long lval;

    lval = (unsigned long)subfunc();
    lval = lval * 0x10000L + subfunc();
    /* ... */;
}
```

〔表2〕評価順序が規定されている演算子

| 演算子 | 説　明 |
|---|---|
| a && b | aを最初に評価して真（非"0"）の場合のみ，bを評価 |
| a \|\| b | aを最初に評価して偽（"0"）の場合のみ，bを評価 |
| a ? b : c | 最初にaを評価して，その結果に応じてbまたはcを評価 |
| a, b | aを評価してその値を捨てて，bを評価 |

A君　：ということはfunc1は引き数が二つで，func2は一つということになるんですか．

N先輩：そういうことだよ．話を戻して，リスト16の関数subfuncは左と右のどちらのsubfuncが先に評価されるかは処理系に依存しているんだ．だから，たまたま，自分の意図している順番で処理される場合は動作するけど，コンパイラがバージョンアップすると誤動作するってことになるんだ．だから，リスト17のように評価順に依存しないようなコーディングを行う必要があるんだよ．

▶ もう一つ式評価順の問題を…

A君　：僕は今まで左から処理されるんだと思ってました．

N先輩：古いC（K&R C）で最適化の性能が低いものだと素直なコードが生成されるから，左から処理されることが多いんだけどね．それじゃ，リスト18の問題点はわかるかい．

A君　：二つのpnt++の「どちらが先に処理されるかわからない」ってことですかね．

N先輩：そうだね．関数の引き数は右からスタックにつまれることは規定されてるけど，関数の引き数の評価順は規定されてないんだ．だから，リスト18のようなプログラムは処理系や最適化の程度によって生成コードが異なる可能性があるから，リスト19のようにコーディングする必要があるんだ．

A君　：はい，わかりました．頭に入れておきます．

### ● 小数点(.)とカンマ(,)

N先輩：リスト20は単純な間違いなんだけど，どこが間違いかわかるかな．

A君　：えーっと…．rdataが実数だから，1.5を1,5に間違えたってことですか．

N先輩：良く気づいたね．小数点（.）をカンマ（,）に間違えた例

〔リスト18〕関数の引き数の評価順（間違い）

```
void subfunc(char *, char *);

char *pnt;

void func(void)
{
    subfunc(pnt++, pnt++);
    /* ... */;
}
```

〔リスト19〕関数の引き数の評価順（正解）

```
void subfunc(char *, char *);

char *pnt;

void func(void)
{
    subfunc(pnt + 1, pnt);
    pnt += 2;
    /* ... */;
}
```

〔リスト20〕小数点(.)とカンマ(,)

```
double rdata;

void func(void)
{
    rdata = 1,5;
}
```

なんだ．小数点(.)をカンマ(,)に間違えても，カンマ演算子とみなされるだけだから，カンマ以降を無視して，整数値が代入されるんだ．これだとBC++ 3.0(Borland C++3.0)は，

```
Warning test.c 5: Code has no effect
                        in function func
```

といった警告を，C++ Builder 6は，

```
警告 W8019 test.c 5: コードは効果を持たない
                              (関数 func)
```

といった警告を出してくれる．でも，VC++ 1.5やVC++ 5.0だと警告レベルを/W4にしても，何も警告が出ないんだ．

A君　：へえーっ，困ったもんですね．

N先輩：リスト20は直接数値を書いてるから，気づく可能性が少しはあるけど，1,5をマクロで定義してたら，警告が出ないと，まず気づかないだろうね．

A君　：そうだと思います．

### ● 整数演算のオーバフロー

N先輩：リスト21は初心者が犯しやすいミスだけど，どこが間違ってるかわかるかい．

A君　：うーむ…なんかあってるように見えますけど．

N先輩：リスト21はキャストが右辺全体にかかってるから，

```
us1 * us2
```

の演算はunsigned shortで行われて，その後unsigned longに零拡張されるんだ．正しくはキャストを，

```
(unsigned long)us1 * us2
```

または，

```
(unsigned long)us1
            * (unsigned long)us2
```

のようにしなければいけないんだ．

A君　：あっ，言われてみればそうですね．

## 関数編
――関数呼び出しの()付け忘れ，内部的に呼び出されているライブラリ関数と同名の関数の定義

### ● 関数呼び出しの()付け忘れ

N先輩：今度はあまりやらないミスだけど，リスト22の間違いがわかるかい．

A君　：これは簡単ですね．funcに()が付いてないじゃないですか．

N先輩：関数呼び出しの，

〔リスト21〕整数演算のオーバフロー

```
unsigned long func(unsigned short us1, unsigned short us2)
{
    unsigned long ul;

    ul = (unsigned long)(us1 * us2);
    return ul;
}
```

〔リスト22〕関数呼び出しの()付け忘れ

```
void func(void);

void mainfunc(void)
{
    func;
}
```

```
func();
```

で()を付け忘れて，

```
func;
```

とするのもコンパイルエラーとならない単純なミスなんだ．普通の人はあまり犯さないミスだと思うけど，僕は最近はもっぱらDelphi(Object Pascal)を使ってて，Object Pascalだと()は付けても付けなくても良いから，たまにC++ BuilderやKylixのC++版なんかを使うと()を忘れてることがあるんだ．このコードは何もしないコードで，たいていのコンパイラではコードが生成されないんだ．見た目はfuncを呼び出しているように見えるから，気づきにくいミスといえるかもしれないね．コンパイラによっては，たとえば，C++ Builder 6だと

```
警告 W8019 test.c 5: コードは効果を
                持たない(関数 mainfunc )
```

のように“warning”が出るけど，VC++のように“warning”レベルが低いと“warning”が出ないコンパイラもあるから注意が必要だね．でも，VC++は“warning”レベルを最高にするとVC++自体のヘッダファイル内の“warning”もけっこう出るから，煩わしいこともあるんだけどね．

A君　：そうですか．

### ● 内部的に呼び出されているライブラリ関数と同名の関数の定義

N先輩：A君はCのライブラリは頭に入ってるかい．

A君　：いいえ，とんでもありません．いつも，マニュアルを見ながらプログラムを組んでいますよ．

N先輩：ANSI Cだとライブラリ関数まできっちりと定めてあるから問題になることは少ないんだけど，古いC(K&R C)だとライブラリの中には別のライブラリを呼び出しているものもあるんだ．たとえば，ライブラリのfread関数がread関数を呼び出してると，プログラム中でread関数を使用していないからといっ

て read 関数を定義してしまうと思ったとおりに動作しなくなるんだ．意識してそんなプログラムを書くことはないと思うけど，使用するコンパイラのライブラリを把握していないと間違って定義してしまうことがあるから注意したほうが良いよ．

だから，他から参照していない関数は(デバッガが static 関数を認識できれば)極力 static 関数として記述するようにしておいたほうが良いかもしれないね．デバッグ時には自分のプログラムを最初に疑ってライブラリやコンパイラにたどり着くまでにある程度の時間がかかるから，原因を見つけるまでにけっこうかかるもんだよ．

A君　：暇なときに，ライブラリ関数を一とおり眺めておくことにします．

## 引き数編
### ――Call by reference の問題点，printf(str)

### ● Call by reference (参照による呼び出し) の問題点

N先輩：今度はリスト 23を見てごらん．

これに main 関数を付けて func1，func2 を呼ぶようにすると，

```
D:¥>test
bdat1 = -1, bdat2 = -1
1                   ← func1のscanfの入力
bdat1 = 1, bdat2 = 0
bdat1 = -1, bdat2 = -1
1                   ← func2のscanfの入力
b1 = 1, bdat1 = 1, bdat2 = -1
```

のような結果になるんだ．func1 は bdat2 を壊しているけど，func2 は正常そうだよね．この現象の説明ができるかい．

A君　：いきなりそんなこと言われてもわからないですよ．

N先輩：それじゃ，簡単に説明してあげよう．scanf で %d と指定すると int データの入力になるんだ．でも，func1 では char データの bdat1 のアドレスを渡しているよね．

A君　：あっ，char は1バイトで，int は4バイト(2バイトのこともあるけど)だから，scanf は渡されたアドレスが int データだと思って書き換えるから，bdat1 の次のアドレスも書き換えてしまうんですね．その結果，func1 の場合は bdat2 が書き換えられちゃうんですよね．

N先輩：大正解．C では Call by value(値による呼び出し)が基本で，そのときは，関数の引き数が一時的なコピーとしてスタックに積まれて渡されるから，関数内で引き数の値を変えても呼び出し元に影響が及ばないんだ．でも，変数のアドレスを渡す Call by reference(参照による呼び出し)も可能だから，変数のサイズを間違えると変な現象に悩まされることになるんだ．func1 は初心者が間違えやすい例の一つなんだよ．それじゃ，func2 は説明できるかい．

A君　：func2 でも scanf に char データを渡してるけど，直接 bdat1 を渡してないから，bdat2 が破壊されないんですね．でも，b1 の次のアドレスが破壊されてるはずですよね．b1 の次のアドレスは定義してないですけど，だいじょうぶなんですか．

N先輩：func2 の場合はたまたま正常に動作するんだ．b1 は自動変数だけど，自動変数はスタック上に割り付けられるっていうことを知ってるかい．

A君　：はい．

N先輩：スタックは通常 int のサイズ(2バイトとか4バイト)単位で割り付けられるんだ．func2 の場合は char データでも int のサイズぶんが割り付けられてるから，b1 の次のアドレスはダミーデータになってるんだ．だから，破壊されても害はないから，正常に動作するんだ．

A君　：でも，b1 以外に自動変数を定義してたらだめですよね．

N先輩：そういうことだね．だから，たまたま動作してるだけなんだよ．

A君　：なんとなくわかりました．

### ● printf(str)――文字列を含むかどうか

N先輩：今度は簡単な問題だ．リスト 24を実行させると，

```
D:¥>test
100
```

となって，100 %と表示されないんだけど，どこが間違ってるかわかるかい．

〔リスト 23〕Call by reference の問題点

```
#include <stdio.h>

char bdat1 = 0;
char bdat2 = 0;

void func1(void)
{
    bdat1 = bdat2 = -1;
    printf("bdat1 = %d, bdat2 = %d¥n",
            bdat1, bdat2);
    scanf("%d", &bdat1);
    printf("bdat1 = %d, bdat2 = %d¥n",
            bdat1, bdat2);
}

void func2(void)
{
    char b1;

    bdat1 = bdat2 = -1;
    printf("bdat1 = %d, bdat2 = %d¥n",
            bdat1, bdat2);
    scanf("%d", &b1);
    bdat1 = b1;
    printf("b1 = %d, bdat1 = %d, bdat2 = %d¥n",
            b1, bdat1, bdat2);
}
```

A君　：えーっと…あっそうか，%はprintfの書式指定になるから，だめなんですね．

N先輩：そうだね．char配列strの内容を表示するのにたいていは，

```
printf(str);
```

としてしまうけど，文字列に%を含んでいると誤動作してしまうんだ．したがって，不具合を避けるには，

```
printf("%s", str);
```

または，

```
fputs(str, stdout);
```

のようにコーディングする必要があるんだよ．

A君　：なんか，すぐにやってしまいそうなことだから，頭にしっかりと入れ込んでおきます．

〔リスト24〕printf(str)

```
#include <stdio.h>

char str[] = "100 % ¥n";

int main(void)
{
    printf(str);
    return 0;
}
```

〔リスト25〕配列のサイズ

```
void func(void)
{
    char ary[11];
    register int n;

    for (n = 0; n <= 11; n++)
        ary[n] = '¥0';
}
```

## 配列編
――配列のサイズ，サイズ指定のある配列の初期化

### ● 配列のサイズ

N先輩：簡単なミスばかり続くけど，リスト25は気が引けるくらい単純なミスだけどわかるかい．

A君　：もちろん，これは簡単ですよ．

```
for (n = 0; n <= 11; n++)
```

が，

```
for (n = 0; n < 11; n++)
```

こうならなくちゃ，いけませんよね．

N先輩：リスト25は，FORTRANやBASICなどの他の言語から移行した人が犯しやすい間違いなんだけど．Cの配列の添字はA君も知ってるように，

```
0〜sizeof(ary) - 1
```

となるから，for文の条件式はn<11とならなくちゃいけないんだ．実行時に範囲チェックをしてくれるコンパイラだと，その時点ですぐにわかる．でも，Cだと実行時の範囲チェックは実行速度の低下につながる関係で範囲チェックがされなくて，添字を超えて自動変数の配列にアクセスするとスタック上に積まれている戻り番地の破壊に結び付くから，簡単に暴走してしまうことになるんだ（といっている僕も久々にFORTRANのプログラムを組んだときに，添字をCの感覚で0から始めてプログラムを暴走させたことがあるんだけど）．

A君　：いろいろな言語を知ってるってことも災いすることがあるんですね．

N先輩：最近，インターネットで騒がれているウィルスの一部はスタック上に積まれている戻り番地を意図的に書き換えて，自分のプログラムを実行させるといったテクニックを使っているものもあるんだよ．

A君　：そうなんですか．やっかいですね．

▶ 境界値問題

N先輩：余談だけど，リスト25は境界値問題の一種といえるかもしれないね．

```
while (++n < nmax)
```

と，

```
while (n++ < nmax)
```

は異なる結果になるから，不等号（<，<=，>，>=）は中級ユーザーでもかなり神経を使う場合があるんだ．デバッグするときには境界値データを使うというのは常識だけど，プログラムを組むときにもアルゴリズムを境界値データで検討してみればバグが減ることになるだろうね．

A君　：はい，わかりました．

N先輩：ところで，自動変数はスタック上に領域が確保されるから，自動変数の配列で配列要素数を越えて書き変えちゃうと，スタックが破壊されて，戻り番地なんかが書き換えられて，暴走してしまうことになるんだ．だけど，リスト25はまったく正常に動作してしまうんだけど，どうしてかわかるかい？

A君　：え〜!!　どうしてです．

N先輩：アセンブリソースのリスト26を見てごらん．arrayの領域をadd esp,-12として12バイトぶん確保しているだろ．

A君　：そうですね．

N先輩：スタックは16ビット用のCコンパイラでも偶数アドレスになるように領域が確保されるから，12バイトになっちゃうんだ．32ビット用のCコンパイラでもintのサイズの整数倍になるように領域が確保されるから12バイトになるんだけどね．実際，リスト25でchar array[12];と書き換えて，アセンブリソースを作成してみるとリスト26とまったく同じになっちゃうんだ．やってみてごらん．

A君　：…そうですね．

N先輩：配列のサイズの11を#defineなんかで定義してると，仕様が変わって，12にした途端に誤動作するってこと

〔リスト 26〕リスト 25のアセンブリソース（C++ Builder 6：-S -O2）

```
    .386p
    ifdef ??version
    if ??version GT 500H
    .mmx
    endif
    endif
    model flat
    ifndef  ??version
    ?debug  macro
    endm
    endif
    ?debug  S "test.c"
    ?debug  T "test.c"
_TEXT   segment dword public use32 'CODE'
_TEXT   ends
_DATA   segment dword public use32 'DATA'
_DATA   ends
_BSS    segment dword public use32 'BSS'
_BSS    ends
DGROUP  group   _BSS,_DATA
_TEXT   segment dword public use32 'CODE'
    align   4
_func   proc    near
?live1@0:
 ;
 ;  void func(void)
 ;
@1:
    add esp,-12    ← arrayの領域を12バイトぶん確保
 ;
 ;  {
 ;      char ary[11];
 ;      register int n;
 ;
 ;      for (n = 0; n <= 11; n++)
 ;
    xor edx,edx
    mov eax,esp
 ;
 ;          ary[n] = '¥0';
 ;
?live1@32: ; EAX = @temp0, EDX = n
@2:
    mov byte ptr [eax],0
    inc edx
    inc eax
    cmp edx,11
    jle     short @2
 ;
 ;  }
 ;
?live1@48: ;
@5:
    add esp,12
    ret
_func   endp
_TEXT   ends
    public  _func
    ?debug  D "test.c" 11979 18498
    end
```

〔リスト 27〕サイズ指定のある配列の初期化

```
char str1[]  = "123456";
char str2[6] = "123456";
```

になるから，原因がわかりにくいこともあるんだよ．

A君　：そういう気がしますね．

### ● サイズ指定のある配列の初期化

N先輩：今度も簡単だけど，リスト 27で str1 と str2 の違いはわかるかい．

A君　：str1 は配列のサイズを省略してて，str2 は配列のサイズを指定してますよね．どう違うんですか．

N先輩：C言語の文字列は最後が '¥0' で終ってるのは知ってるよね．

A君　：はい．

N先輩：だから，"123456"は 123456の文字列と '¥0' のことなんだけど，str1 にはこれらがそのまま初期値としてセットされて配列のサイズは 7バイトになるんだ．

A君　：ということは，str2 は配列のサイズを 6と指定してるから最後の '¥0' がセットされないってことですか．

N先輩：そういうことだね．

A君　：だったら，

```
char str[5] = "123456";
```

としたら 12345の文字列がセットされるんですか．

N先輩：それがね，その場合はコンパイラが初期化データが多すぎるって"error"を出すと思うよ．コンパイラによっては"warning"のこともあると思うけど．

A君　：それって，どうしてなんですか．

N先輩：たとえばね，16進文字列を定義するときに，

```
char str[] = "0123456789ABCDEF";
```

とすると最後に余分な '¥0' が付加されるだろ．

A君　：はい．

N先輩：こういうときは，

```
char str[16] = "0123456789ABCDEF";
```

とすれば余分な '¥0' が付加されないから，こういった使い方を想定しているんだと思うよ．

A君　：あっ，そうなんですか．

## ポインタ編

――自動変数へのポインタを返す，ポインタ同士の減算，配列とポインタ，ポインタの初期化，freeで解放した直後の領域の参照，Null pointer assignment

N先輩：ポインタは Cでもっとも難解なものの一つだけど，

```
int *pointer;
```

とすると pointer はポインタ型変数で，この中にポインタ値が格納されていて，特定のアドレスを指し示しているんだ．たとえば，

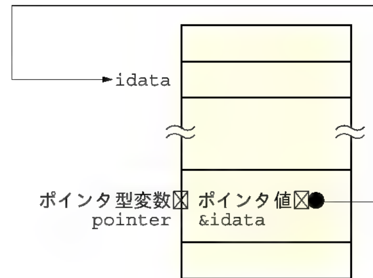
```
int idata;
pointer = &idata;
```

とすると図 1のような感じになるんだ．

A君　：これくらいまでならなんとなくわかってるつもりなんですけど．

N先輩：普通はポインタ型変数とポインタ値はどっちもポインタと呼んでいて，文脈の中でどっちか判断する記述になってることが多いんだ．

〔図1〕ポインタ型変数とポインタ値




A君　：そうですよね．

● 自動変数へのポインタを返す

N先輩：リスト28は文字列を返す関数だけど，間違いがあるんだ．どこが間違ってるかわかるかい．

A君　：うーむ…なんとなくあってる気がしますけど．

N先輩：それじゃ，リスト28にmain関数を追加して，printfでfunc関数が返す文字列を表示させてごらん．

A君　：はい…あれっ，何も表示されないですね．

▶ 自動変数の落とし穴

N先輩：func関数のbufは自動変数になってるだろ．

A君　：はい．

N先輩：自動変数はその関数に入った時点でスタック上に領域が確保されて，その関数を抜け出るとその領域が自動的に解放されてしまうんだ．だから，func関数が返すポインタが実体のないところを指していることになるんだ．C++ Builder 6だと，

　　警告 W8075 test.c 10:問題のあるポインタの変換（関数 func ）

という"warning"を出してくれるから気づくけど，"warning"レベルが低いと出ないコンパイラもあるからね．

A君　：へえー，そうだったんですか．ということはどうすればいいんですか．

▶ スタティック変数にすれば…

N先輩：いちばん簡単な対策は

```
static char buf[20];
```

とすることだよ．staticにしておけばプログラム終了時まで領域が確保されてるからね．

A君　：でも，そうすると次に呼び出したときに前の内容が壊れちゃいますよね．

N先輩：前の値を壊したくないときは，func()で返された文字列を別バッファにコピーしておけばいいんだよ．

A君　：あっ，そうですね．

N先輩：もっとも，func()関数内でmallocで領域を確保してそのアドレスを返すという方法もあるんだけどね．でも，この方法はどこで領域を解放するかって問題があるから，必ずしも良い方法だとはいえないんだけど．

〔リスト28〕自動変数へのポインタを返す

```
#include <stdio.h>

unsigned count;

char *func(void)
{
    char buf[20];

    sprintf(buf, "counter %u", ++count);
    return buf;
}
```

〔リスト29〕ポインタ同士の減算

```
long ldat[10];

long func(long *pl1, long *pl2)
{
    return pl1 - pl2;
}

void mainfunc(void)
{
    long ldif = func(&ldat[1], &ldat[0]);
}
```

A君　：そうですね．

● ポインタどうしの減算

N先輩：リスト29はポインタ同士の減算を行ってるプログラムだけど，ldifは何になるかわかるかい．

▶ ポインタの加減算のサイズは？

A君　：えっと…longの配列の1番目と0番目の差だから4ですかね．

N先輩：ポインタの加減算がポインタのサイズで行われることは知ってるよね．

A君　：はい．

N先輩：たとえば，

```
char *pc;
```

だとpc + 1は1バイトの加算となって，

```
long *pl;
```

だとpl - 1は4バイトの減算となるんだ．

A君　：そうですね．それくらいならわかってます．

N先輩：そうか，意外と優秀なんだ（おっと失礼！）．ポインタ同士の減算でも同じことなんだ．charへのポインタ同士を減算する場合は迷うことはないけど，short，long，構造体などのポインタ同士を減算する場合はポインタ間のバイト数じゃなくて，

　　バイト数 / ポインタサイズ

が得られるんだ．リスト29だと，

```
pl1 - pl2
```

は，

```
((char *)pl1 - (char *)pl2) /
                        sizeof(long)
```

と同じことになるんだ．当たり前のことなんだけど，

僕はCの中級者になったころに間違えたことがあるよ．

A君　：僕も良く頭に入れておきます．

## ● 配列とポインタ

N先輩：リスト30のfunc1とfunc2はどう違うかわかるかい．

▶ 関数の引き数だとポインタと配列が同じになる？

A君　：片方が，ポインタでもう一方が配列ということじゃないんですか．

N先輩：実はね，どちらも同じことになるんだ．Cは関数の引き数に配列を一括して渡すことはできなくて，配列風に宣言してもポインタになるんだ．

A君　：どうしてです？

N先輩：わかりやすく説明できないな…．引き数で，

```
func(array);
```

のように配列を渡してもCコンパイラは，

```
func(&array[0]);
```

のように最初の要素のアドレスと解釈してしまうんだ．言いたいことは，"配列"は定数で"ポインタ"は変数なんだけれど，関数の引き数はスタック経由で渡される変数だから，配列風に宣言してもポインタと同じことになるんだよ．わかるかなぁ．

A君　：なんか，わかったようなわからないような．

▶ 外部変数だとポインタと配列は…？

N先輩：それじゃ，混乱ついでにもっと混乱させることになるけどリスト31(a)とリスト31(b)は片方で定義して，もう一方で参照してるけど，どこが間違ってるかわかるかな．

A君　：そりゃ，配列で定義した変数をポインタとして参照してるところでしょ．

N先輩：良くわかってるじゃない．関数の引き数だと配列風の宣言とポインタ形式の宣言が同じように扱われるから，配列とポインタを同じものと錯覚して間違える場合があるけど，外部変数だと，

```
extern char *ptr;
```

と，

```
extern char ptr[];
```

では違ったコードが生成されるんだ(違いがわからない人はアセンブルリストを生成してよく眺めること)．僕もCを始めたころに間違えた経験があるよ．この問題に限らず，ポインタはトラブルメーカーだから，気をつけないといけないだろうね．僕も*が三つも四つもと付いたり，関数のポインタのポインタとかいったプログラムに出会うと頭がパニックになることがあるよ．自分が書くときはそうでもないんだけどね．

A君　：ポインタは難しいですよね．

## ● ポインタの初期化

N先輩：リスト32のinitpnt関数はポインタを初期化するつもりなんだけど，func関数のように呼んでも引き数piは変化しないんだ．なぜだかわかるかい．

A君　：良さそうに見えますけど，…あっ，さっき説明してもらいましたけど，CはCall by value(値による呼び出し)が基本だから，ポインタ型変数piの中身のポインタ値が引き数としてスタックに積まれて渡されるから，関数initpnt内で引き数の値を変えても呼び出し元に影響が及ばないんですよね．

N先輩：そうだね．だったら，うまく動くようにするにはどうすれば良いかわかるかい．

〔リスト30〕配列とポインタ(その1)

```
void func1(char *p)      /* ←こちらはポインタで */
{
    /* ... */;
}

void func2(char p[])     /* ←こちらは配列? */
{
    /* ... */;
}
```

〔リスト31〕配列とポインタ(その2)

```
char ptr[100];
```

(a)

```
extern char *ptr;

void func(void)
{
    *ptr = '¥0';
}
```

(b)

〔リスト32〕ポインタの初期化(間違い)

```
#include <stdlib.h>

void initpnt(int *pi)
{
    pi = malloc(10 * sizeof(int));
}

void func(void)
{
    int *pi;

    initpnt(pi);
}
```

〔リスト33〕ポインタの初期化

```
#include <stdlib.h>

void initpnt(int **pi)
{
    *pi = malloc(10 * sizeof(int));
}

void func(void)
{
    int *pi;

    initpnt(&pi);
}
```

〔リスト34〕freeで解放した直後の領域の参照（間違い）

```
#include <stdlib.h>

struct list_t {
    struct list_t *pnext;
    int data1;
    /* ..... */
    int datan;
};

void free_list(struct list_t *plist)
{
    while (plist) {
        free(plist);
        plist = plist->pnext;
    }
}
```

〔リスト35〕リスト34の修正版

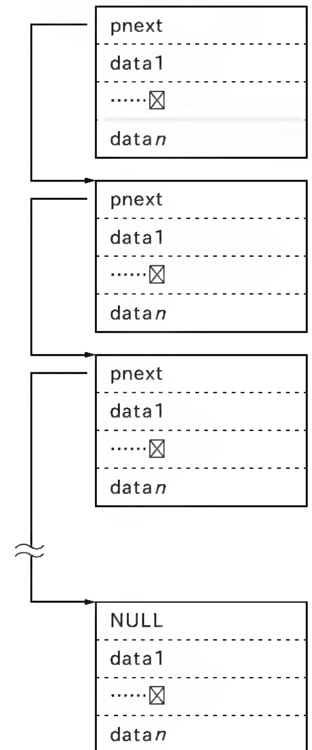
```
#include <stdlib.h>

struct list_t {
    struct list_t *pnext;
    int data1;
    /* ..... */
    int datan;
};

void free_list(struct list_t *plist)
{
    while (plist) {
        struct list_t *pnext;

        pnext = plist->pnext;
        free(plist);
        plist = pnext;
    }
}
```

〔図2〕リスト構造




A君　：えっーと，…ポインタ型変数piのアドレスを渡せば良いんだから，リスト33のようにすれば良いんじゃないですか．

N先輩：だいぶわかってきたみたいだね．

● freeで解放した直後の領域の参照

N先輩：今度はfreeの間違いを見つけてもらおう．リスト34はどこが間違ってると思う？

▶ リスト構造とポインタ

A君　：リスト34は何をしてるのかいまひとつわからないですけど．

N先輩：図2のように次の構造を指し示すポインタとデータ要素から構成されるデータ構造をリスト構造って言うんだ．最初にmallocで確保した領域のアドレスをpnextに代入して次々にチェーンさせておくと先頭から次々にたどれるだろ．

A君　：はい．

N先輩：最後はpnextにNULLを入れておくことで判定するんだ．

A君　：だったら，

```
while (plist)
```

は，

```
while (plist != NULL)
```

ってしなくちゃいけないんじゃないですか．

N先輩：そうしたほうがわかりやすいかもしれないね．でも，どちらでもかまわないんだよ．たとえば，intのデータidataで，

```
if (idata)
```

ってするのと，

```
if (idata != 0)
```

ってするのは同じことだろ．

A君　：はい．

N先輩：ポインタでもこれと同じでどちらでもいいんだよ．

A君　：そうなんですか．

N先輩：それで，リスト34はリスト構造で確保した領域をすべて解放するプログラムのつもりなんだ．

A君　：む…．そこまで説明してもらっても，どこが間違ってるのかわからないです．

N先輩：リスト34はfreeで解放した構造体に残ってる次へのポインタを使ってるだろ．

A君　：はい．

N先輩：freeで解放した領域ってのはその中身が残ってるかどうかわからないんだ．ほとんどの場合，解放前の値が残っていて正常に動作すると思うけど．マルチタスクのOSだとfreeで解放した直後に別タスクが実行されて，解放した領域を別の用途に再確保して使ってるという可能性もあるんだ．

A君　：だったら，どうすればいいんですか．

N先輩：リスト35のように次アドレスを一時変数に覚えさせておけばいいんだよ．

A君　：あっ，そうですね．

● Null pointer assignmentとは

N先輩：A君はNull pointer assignmentって知ってるかい．

A君　：いいえ．でも，この間，そういうメッセージが出たんですけど，プログラムは動いてたから，無視しちゃいました．もしかして，無視しちゃいけなかったんですかね．

N先輩：えっ！　君!!　無視したのか？　コンパイラの警告にはそれなりに敬意を払わないとプログラマ失格だぞ！

A君　：えっ?!　すみません….

▶ 軽視できないメッセージ

N先輩：む…．半分は冗談だけれど，半分は本気だよ．Null pointer assignmentは初期化していないポインタを使用した場合にプログラム実行終了時に表示されるんだ．たとえば，リスト36でポインタpointerに初期値をセットしないでfuncを実行させるとプログラム終了時にNull pointer assignmentと表示されるはずだよ．初心者の場合，このメッセージの意味がわからないから，プログラムがほとんど動いていると無視してしまうことが多いんだけど，デバッグの仕方が悪いから，動いていると思うだけなんだよ．

N先輩：Null pointer assignmentはCプログラムでは変数が割り付けられることがない0番地付近に書き込みを行った場合に表示されるんだ．初期化していない静的変数のポインタでリスト36のように書き込みを行うとスタートアップルーチンで変数領域が0クリアされているから，0番地に書き込んだことが検出できるんだ．スタートアップルーチンではプログラム終了時に0番地付近を調べて破壊されていればNull pointer assign mentのメッセージを表示するんだ．

▶ そのまま動かせば…

N先輩：32ビットのOSだとメモリプロテクション機能をもっているから，実行時に文句をいわれてタスクが停止させられてしまって簡単に検出できるけど，MS-DOSのようにメモリプロテクション機能をもっていない場合には，破壊している箇所の検出はけっこうめんどうなんだ．ハードウェアブレークをサポートしているデバッガが使用できれば，0番地付近にライトプロテクトをかけておけば簡単に検出できる．でも，もし使用できなければタイマ割り込みなどで定期的に0番地を調べるプログラムを走らせておくといった対策が必要となるんだ．CodeViewやTurboDebuggerなどのウォッチポイントで調べるという方法もあるけど，ソフトウェアで実現されている場合には検出までに数時間かかることも珍しくないんだ．

A君　：僕はたいへんなことを無視しちゃったみたいですね．

N先輩：そのプログラムは後で初期化していないポインタを使っている箇所を探してごらん．

A君　：はい，そうします．

N先輩：余談だけど，ヌルポインタとヌルストリングの違いはわかるかい．

A君　：えっ，ヌルポインタは今でてきたばかりだから，0のことですか．

N先輩：それじゃ，リスト37を見てごらん．これは空の文字列をコピーするプログラムのつもりで書いたんだけど，どれが合ってるかわかるかい．

A君　：えーっと，…func3ですよね．

N先輩：そうだね，func3が正解だね．NULLは一応，NULLというマクロが定義されてて，NULLの定義を見てみるとコンパイラによっては，

```
#define NULL 0
```

って書いてあることがあるから〔ポインタということで#define NULL ((void *)0)と定義している処理系もある〕，0のことだと思っても大きな間違いじゃないと思うけど．NULLはポインタに対する特別な値で，通常はどこも指していない無効なポインタとして使われるんだ．だから，アドレス演算子&を適用した結果がヌルポインタになったり，ライブラリ関数の成功した返り値がヌルポインタになったりすることがないことが保証されているんだ．ヌルストリングは空の文字列のことで，具体的には

```
""
```

のことなんだよ．

A君　：へえー，そうなんですか．

N先輩：だから，func1は無効なNULLポインタからコピーしてるから間違いで（図3），func2の'¥0'は文字コードの0のことだから，同じく間違ってるんだ．普通に文字列をコピーするんだったら，

```
strcpy(buf, "str");
```

ってな感じで間違う人はいないと思うけど，空の文字列のコピーだと，初心者は理解が浅いから間違えてしまうことがあるんだ．

A君　：わかりました．

〔リスト36〕初期化していないポインタ

```
char *pointer;

void func(void)
{
    *pointer = 'A';
}
```

〔リスト37〕空の文字列のコピー

```
#include <string.h>

char buf[1024];

void func1(void)
{
    strcpy(buf, NULL);
}

void func2(void)
{
    strcpy(buf, '¥0');
}

void func3(void)
{
    strcpy(buf, "");
}
```

## 文字列編

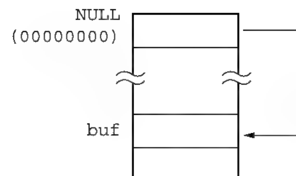
――文字列の比較，文字列用の領域の確保，同じ文字列へのポインタ

### ● 文字列の比較

N先輩：リスト38も初心者に多い間違いだけど，どこが間違ってるかわかるかい．

A君　：えーと…あっ，文字列の比較なのにポインタのアドレ

〔図3〕strcpy( buf, NULL);




スを比較してますよね．

N先輩：そういうことだね．だったら，書き直したら，どうなるかな．

A君　：えーと…．リスト39のようになるんじゃないですか．

N先輩：A君はC言語の文字列をちゃんと理解してるみたいだね．

● 文字列用の領域の確保

N先輩：今度はリスト40とリスト41はどちらかが間違っているんだけど，どこが間違ってるかわかるかな．

▶ 文字列のコピーのときは"¥0"のことを忘れない

A君　：ふーむ…どっちも合ってる気がするけど．

N先輩：実はリスト40の，

```
p = malloc(strlen(pnt));
```

は，

```
p = malloc(strlen(pnt) + 1);
```

としないといけないんだ．

A君　：あっ，そうですね．文字列のコピーは最後に'¥0'がコピーされるから+1しないといけないんですよね．

N先輩：それじゃ，リスト42はどこがいけないと思う？

A君　：えっーと…．あっ，sizeof(pnt)はchar *pntのサイズを返すだけだから，間違ってますよね．

N先輩：そうだね．文字列用の領域の確保で最後の'¥0'のサイズを忘れて領域を確保するといった間違いはよくあるパターンだね．リスト41のsizeofだと'¥0'のサイズを含んだ大きさが返されるから，同じプログラムでsizeofとstrlenを使い分けているときに間違いやすい状態になるよ．その関連で，リスト42のような間違いも犯すことがあるんだ．この間違いを犯しても，直後の変数を破壊するだけだから，しばらくは正常に動作して，破壊された変数をアクセスする個所で不具合が発生したり，プログラム終了時に暴走するとかいう現象になることもあるから，なかなか間違えた個所にたどり着かなくて困ることもあるんだ．

A君　：気をつけます．

N先輩：それじゃ，リスト43を見てごらん．リスト42の引き数を配列風に宣言してサイズまで指定してあるんだけど，これはどうなるかわかるかい．

A君　：char *pnt[10]としてあるんだから，sizeof(pnt)は10を返すんじゃないんですか．

N先輩：さっきも言ったけど，関数の引き数を配列風に宣言し

〔リスト38〕文字列の比較(間違い)

```
int strequ(char *p1, char *p2)
{
    if (p1 == p2)
        return 1; /* 等しい */
    else
        return 0; /* 等しくない */
}
```

〔リスト39〕文字列の比較(正解)

```
#include <string.h>

int strequ(char *p1, char *p2)
{
    if (strcmp(p1, p2) == 0)
        return 1;   /* 等しい */
    else
        return 0;   /* 等しくない */
}
```

〔リスト40〕文字列用の領域の確保(strlen編：間違い)

```
#include <string.h>
#include <stdlib.h>

void func(char *pnt)
{
    char *p;

    p = malloc(strlen(pnt));
    if (p != NULL) {
        strcpy(p, pnt);
    }
    /* ... */;
}
```

〔リスト41〕文字列用の領域の確保(sizeof編)

```
#include <string.h>
#include <stdlib.h>

void func(void)
{
    static char pnt[] = "abcdefgh";
    char *p;

    p = malloc(sizeof(pnt));
    if (p != NULL) {
        strcpy(p, pnt);
    }
    /* ... */;
}
```

〔リスト42〕文字列用の領域の確保(sizeof編：間違い)

```
#include <string.h>
#include <stdlib.h>

void func(char *pnt)
{
    char *p;

    p = malloc(sizeof(pnt));
    if (p != NULL) {
        strcpy(p, pnt);
    }
    /* ... */;
}
```

〔リスト43〕文字列用の領域の確保(sizeof編：間違い)

```
#include <string.h>
#include <stdlib.h>

void func(char pnt[10])
{
    char *p;

    p = malloc(sizeof(pnt));
    if (p != NULL) {
        strcpy(p, pnt);
    }
    /* ... */;
}
```

てもポインタになるから，1次元の場合はサイズを指定しても意味がないんだ．

A君　：なんか変な感じがしますね．

N先輩：混乱の元だから，止めたほうが良いだろうね．

A君　：そうですね．

N先輩：余談だけど，sizeof(char)の値はわかるよね．

A君　：そりゃ，1ですよ．

N先輩：だったら，sizeof('a')の値はなんになるかわかるかい．

A君　：それも1じゃないんですか．

N先輩：それが，違うんだよ．実際にprintfで表示してみてごらん．

A君　：あっ，2って表示されましたね．

N先輩：Cの文字定数はint型をもつから，sizeof('a')は16ビットコンパイラだと2が返されるし，32ビットのコンパイラだと4が返されるんだよ．

A君　：へえーっ，知りませんでした．

N先輩：C++だと，また違った結果になるんだけど，混乱の元だから，止めておくよ．

### ● 同じ文字列へのポインタ

N先輩：それじゃあね，リスト44を見てごらん．ポインタp1の指し示す先頭を，

```
*p1 = '1';
```

のように変更したとき，ポインタp2の指し示す先頭も書き換えられてしまうってことがあると思うかい．

▶ 同じ文字列へのポインタの片方の先頭を変更するともう片方も…

A君　：p2 = p1のようにしなくてもですか．

N先輩：そうだよ．

A君　：だったら，あり得ないんじゃないですかね．

〔リスト44〕同じ文字列へのポインタ（間違い）

```
char *p1 = "ABC";
char *p2 = "ABC";

void func(void)
{
    *p1 = '1';
    if (*p2 == 'A')
        /* ... */;
    else
        /* ... */;
}
```

〔リスト45〕構造体の隙間

```
struct {
    unsigned char  uc;
    unsigned short us;
    unsigned long  ul;
} s1;

struct {
    unsigned long  ul;
    unsigned short us;
    unsigned char  uc;
} s2;
```

N先輩：それがね．最適化がからむとそういう現象が起きることがあるんだよ．

A君　：どういうことですか．

N先輩：リスト44のように同じ文字列へのポインタがあると最適化の結果，

```
"ABC"
```

という文字列が一つしか生成されずに，ポインタp1とp2が同じ番地を指し示すようなコードを生成する処理系があるんだよ．最適化のコンパイルスイッチでそれを活かしたり，無効にしたりできるんだけどね．ANSI Cだと基本的に文字列の変更はできないから，そういう最適化のコードもANSI Cに準拠していることになるんだ．その結果

```
*p1 = '1';
```

のように変更すると*p2の内容も変わってしまうことになるんだ．リスト44のように並んでいる場合はすぐに気づくけど，関数内でstatic変数として割り付けているものなんかがこの現象になるとなかなか気づかないと思うよ．

A君　：そうですね．

N先輩：処理系によっては，

```
char *p1 = "ABC";
```

とした場合，文字列"ABC"は書き込み禁止の領域に生成されることもあるから，

```
*p1 = '1';
```

という書き込みができない可能性もあるし．コンパイラが生成する書き込み禁止の領域が実際に書き込みできないかどうかはターゲット次第だけどね．書き換えたい場合はポインタではなく，配列として，

```
char p1[] = "ABC";
```

とする必要があるんだ．

## 構造体編
――構造体の隙間，構造体の比較，構造体のコピーと配列のコピー，共用体の初期化

### ● 構造体の隙間

N先輩：今度は少し頭の体操になるかもしれないけど，リスト45の構造体s1のサイズはいくつになるかわかるかい．

A君　：charが1バイト，shortが2バイト，longが4バイトとしたら，7バイトになるんじゃないですか．

N先輩：半分正解．7バイト以外になることもあるんだけど，何バイトになるか説明できるかい．

A君　：うーむ…わからないです．

N先輩：CPUによって大きく違ってくるんだけど，86系だと8バイトになることがあるんだ．

A君　：どう計算したら8バイトになるんですか．

N先輩：8086のような16ビットCPUだとshortやlongの変

数のアクセスは偶数アドレスから行ったほうが速くなるんだ．だから，16ビット版のVC++でオプションスイッチを指定しないと

```
struct {
    unsigned char  uc;  /* 1 バイト */
                        /* 1 バイト */
    unsigned short us;  /* 2 バイト */
                        /* 0 バイト */
    unsigned long  ul;  /* 4 バイト */
} s1;    /* sizeof(s1)    8      */
```

という感じで各メンバが偶数アドレスから始まるようにメンバ間に隙間調整のダミーのバイトが入れられるんだ（図4）．この隙間調整を止めるには/Zpというオプションスイッチを指定すればいいんだけど．こうすればA君のいうように7バイトになるんだ（図5）．32ビットCPUだと4の倍数のアドレスから行ったほうが速くなって，最近のPentiumプロセッサは8の倍数のアドレスから行ったほうが速くなるから，C++ Builder 6だと8バイト境界がデフォルトの設定になってるんだ．コンパイラによって，違った結果になる可能性もあるけど，リスト45の構造体だとたぶん同じ結果なると思うよ．

A君　：へえー，そうなんですか．

N先輩：それじゃ，リスト45の構造体s2だとどうなるか説明できるかい．

A君　：構造体s2はパックしても，しなくても7バイトじゃないんですか．

N先輩：実はね，構造体をパックしない場合は最後にダミーの1バイトが入れられて8バイトになるんだよ．

A君　：なんかややこしいですね．

N先輩：構造体の代入ができるのに比較ができない理由がここらあたりにあるんだよ．sizeofで返されるサイズは構造体の隙間を含んだサイズとなるから，自分でいろいろな構造体のバイト数を調べてみれば良いと思うよ．

A君　：はい，そうします．

N先輩：CPUの種類が変わるとこの辺のところも少し変わってきて，68000だと奇数番地からのワード/ロングワードアクセスが許されないから，構造体がパックされることはなくなってリスト45の構造体のサイズはかならず8バイトになるんだ．32ビットCPUの中にはかならずロングワード単位でしかアクセスできないものもあるから，そういうCPUだと，

```
struct {
    unsigned char  uc;  /* 1 バイト   */
                        /* 3 バイト   */
    unsigned short us;  /* 2 バイト   */
                        /* 2 バイト   */
    unsigned long  ul;  /* 4 バイト   */
} s1;    /* sizeof(s1)   12        */
```

〔図4〕構造体s1のメモリ割付け

| | |
|---|---|
| +0 | uc |
| +1 | ダミー☒ |
| +2 | us |
| +3 | |
| +4 | u1 |
| +5 | |
| +6 | |
| +7 | |

〔図5〕構造体s1のメモリ割付け（/Zp）

| | |
|---|---|
| +0 | uc |
| +1 | us |
| +2 | |
| +3 | u1 |
| +4 | |
| +5 | |
| +6 | |

〔図6〕構造体s1のメモリ割付け（ロングワード境界）

| | |
|---|---|
| +0 | uc |
| +1 | ダミー☒ |
| +2 | |
| +3 | |
| +4 | us |
| +5 | |
| +6 | ダミー☒ |
| +7 | |
| +8 | u1 |
| +9 | |
| +10 | |
| +11 | |

というような感じになるはずだよ（図6）．

A君　：ますますこんがらがってきましたよ．

N先輩：といっても，プログラム内で独自に定義する構造体だととくに問題になることはないんだけど，

- CP/M-68Kのファイルフォーマット
- 68K系のバスエラーのスタックフレーム

のようにすでに決められているフォーマットを構造体で表現するときに問題となるんだ．たとえば，CP/M-68Kのファイルフォーマットを定義するリスト46はintが32ビットのコンパイラだと隙間が生じてしまうから，リスト47のように定義しなければいけなくなるんだ．リスト47だと簡単に演算ができなくなるから，プログラムがめんどうになるんだけど，仕方がないんだよね．

A君　：僕も68K系をやるようになったら，気をつけないといけませんね．

N先輩：余談だけど，構造体のメンバのオフセット位置を求めるためにANSI Cではヘッダファイル<stddef.h>でoffsetofというマクロが定義されるようになったんだ．これは古いC（K&R C）でも，たいてい利用できるから，必要な場合はマクロ定義すれば使えると思うよ

〔リスト46〕CP/M-68Kファイル・ヘッダ

```
struct s68kheader_t {
    unsigned short magic;
/* ヘッダ 0x601a : text, data, bss 連続 */
                     /*    0x601b : text, data, bss 非連続 */
    unsigned long tsize;   /* テキスト セグメント バイト数      */
    unsigned long dsize;   /* データ    セグメント バイト数      */
    unsigned long bsize;   /* BSS      セグメント バイト数      */
    unsigned long ssize;   /* シンボル テーブル バイト数        */
    unsigned long stksize; /* イニシャル スタック サイズ        */
    unsigned long tstart;  /* テキスト セグメント開始アドレス   */
    unsigned short freloc; /* リロケーション フラグ            */
    unsigned long dstart;  /* データ セグメント開始アドレス     */
    unsigned long bstart;  /* BSS     セグメント開始アドレス    */
} s68kheader;              /* CP/M-68K ファイル・ヘッダ        */
```

〔リスト47〕CP/M-68Kファイル・ヘッダ

```
struct s68kheader_t {
    unsigned short magic;
                          /* ヘッダ 0x601a : text, data, bss 連続   */
                          /*     0x601b : text, data, bss 非連続    */
    unsigned short tsize[2];  /* テキスト セグメント バイト数       */
    unsigned short dsize[2];  /* データ    セグメント バイト数      */
    unsigned short bsize[2];  /* BSS      セグメント バイト数       */
    unsigned short ssize[2];  /* シンボル テーブル バイト数         */
    unsigned short stksize[2];/* イニシャル スタック サイズ         */
    unsigned short tstart[2]; /* テキスト セグメント開始アドレス    */
    unsigned short freloc;    /* リロケーション フラグ             */
    unsigned short dstart[2]; /* データ セグメント開始アドレス      */
    unsigned short bstart[2]; /* BSS     セグメント開始アドレス     */
} s68kheader;                 /* CP/M-68K ファイル・ヘッダ         */
```

〔リスト48〕offsetofマクロの例

```
#ifdef  __STDC__
#include <stddef.h>
#else
#ifndef offsetof
typedef unsigned int size_t;
#define offsetof(s_name, member)
(size_t)&(((s_name*)0)->member)
#endif
#endif

struct st {
    int i1;
    int i2;
    int i3;
} s;

void func(void)
{
    unsigned offset = offsetof(struct st, i3);
}
```

〔リスト49〕構造体の比較(間違い)

```
#include <string.h>

struct sample {
    unsigned char  uc;
    unsigned short us;
    unsigned long  ul;
} smp1, smp2;

int cmp(struct sample *psmp1, struct sample *psmp2)
{
    return memcmp(&psmp1, &psmp2, sizeof(struct sample));
}
```

〔リスト50〕構造体の比較

```
struct sample {
    unsigned char  uc;
    unsigned short us;
    unsigned long  ul;
} smp1, smp2;

int cmp(struct sample *psmp1, struct sample *psmp2)
{
    return (psmp1->uc == psmp2->uc
           && psmp1->us == psmp2->us
           && psmp1->ul == psmp2->ul) ? 0 : 1;
}
```

〔リスト51〕配列のコピー

```
#include <string.h>

struct sample {
    char ary[100];
} smp1, smp2;

void func(void)
{
    memcpy(&smp1, &smp2, sizeof(smp1));
}
```

(リスト48).

A君　：はい,頭に入れておきます.

### ● 構造体の比較

N先輩：それじゃ,間違い探しをやってみよう.リスト49は構造体を比較するプログラムだけど,間違ってるんだ.どこだかわかるよね.

A君　：えっ,合ってるんじゃないですか.

N先輩：もう一度,図4や図6を見てごらん.

A君　：あっ,構造体の隙間のダミーのところの初期値が違ってたら,memcmpで比較できないですよね.

N先輩：そういうことだね.だから,構造体の比較はめんどうでもリスト50のようにメンバ同士の比較をしないといけないんだ.もっとも,構造体を定義したときにかならず0クリアしておくようにすればmemcmpで比較しても問題なくなるけどね.

### ● 構造体のコピーと配列のコピー

N先輩：今度はひとつ頭の体操をしてみよう.配列をコピーするときに,

```
char ary1[100], ary2[100];
ary1 = ary2;
```

というような具合にコピーできればいいんだけど,だめだよね.この配列をもっとも効率的にコピーする方法を考えてごらん.

A君　：リスト51のようにmemcpyを使えばいいんじゃないですか.

N先輩：memcpyよりももっと効率の良い方法があるんだよ.

A君　：そんなこといわれてもわかんないですよ.

N先輩：それじゃ,ヒントを一つ出してあげよう.構造体は

```
struct sample smp1, smp2;
smp1 = smp2;
```

というような具合にコピーができることは知ってる

かい.

A君　：さっき言われましたよね.

N先輩：これを応用すればいいんだよ.

A君　：うーむ…あっ!! 配列を構造体にしてしまえば良いんですね. リスト52のようにすれば良いんですよね.

N先輩：ピンポン，ピンポン，大正解. memcpyを使う場合は効率はmemcpy次第になるけど，構造体にしてコピーすればコンパイラがコンパイルする時点でコピーするサイズなんかがわかっているから，より効率の良いコードに落ちるんだ. それに，プログラムも読みやすい方向になるしね. 構造体のコピーは通常の最適化レベルでインライン展開してくれるよ. もっとも，最適化のレベルを上げるとmemcpyもインライン展開されて同じコードに落ちるけどね. intが4バイトの68K系のコンパイラだと，さっきいったようにダミーのバイトが挿入されて構造体のサイズが4の倍数に調整されてロングワード単位でコピーしてくれるんだ.

A君　：これは良いことを教わりました. 覚えておいて何かの機会に使ってみることにします.

### ● 共用体の初期化

N先輩：話はちょっと変わるけど，共用体は知ってるよね.

A君　：知ってますけど.

N先輩：共用体の初期化はどうするか知ってるかい.

A君　：いや….

N先輩：ANSI Cだと共用体の最初のメンバで初期化できるんだ(リスト53).

A君　：へえーっ，知りませんでした.

## マクロ編

――マクロの副作用，マクロ定義時の問題，マクロと型定義，TRUE, FALSE

### ● マクロの副作用

N先輩：マクロの副作用もよく知られているよね.

A君　：はい.

N先輩：たとえば，

```
#define    toupper(c)
    ((islower(c)) ? _toupper(c) : (c))
```

のようなマクロだと，

```
toupper(*p++)
```

とすると*p++が2度評価されてしまうから，気をつけないといけないんだ. ANSI Cだとヘッダファイル内で定義されているマクロに関して，このような副作用があってはいけないことになっているから，副作用が生じる場合は関数で定義されているんだ. でも，ANSI C完全準拠のモードでないとこの仕様にならないコンパイラもあるから，安心はできないけどね. マクロ定義時に()を多用する理由もわかってるよね.

〔リスト52〕配列のコピー

```
struct sample {
    char ary[100];
} smp1, smp2;

void func(void)
{
    smp1 = smp2;
}
```

〔リスト53〕共用体の初期化

```
union udata_t {
    long ldata;
    char cdata[4];
} udata = {0x12345678L};
```

〔リスト54〕マクロのトラブル(その1)

```
#define macro(x, y) x = 1;  y = 2;

int flag, xval, yval;

void func(void)
{
    if (flag)
        macro(xval, yval);
}
```

A君　：はい. たとえば，

```
#define    abs(x)    x>=0 ? x : -x
```

という感じで定義して，

```
abs(x - y) + 1
```

で呼び出すと，

```
x - y>=0 ? x - y : -x - y + 1
```

と展開されるから，まずいのでは? だから，引き数や全体を( )でくくって，

```
#define    abs(x)
                 ((x)>=0 ? (x) : -(x))
```

という感じで定義しないといけないんですよね.

N先輩：そうだね. マクロに関するトラブルはこの他にもきりがないけど，たとえば，リスト54の間違いはわかるかい.

A君　：マクロが展開されると，

```
if (flag)
    xval = 1;
yval = 2;
```

となるのがまずいですよね.

N先輩：リスト54だとソースリストを眺めてもマクロが頭に入っていないとなかなか見つけにくいと思うけど. この場合はマクロを，

```
#define  macro(x, y)  (x = 1, y = 2)
```

または，

```
#define macro(x, y)  {x = 1; y = 2;}
```

のように記述して単一の文(ブロック)として処理されるようにしておかないといけないんだ.

A君　：はい，頭に入れておきます.

N先輩：話は変わるけど，初心者は#defineでマクロを定義するときに最後に不要な;を付けてしまうことがあるから，注意が必要だよ. リスト55のようにコンパイルエ

ラーが出れば良いけど，出ないような箇所で使用されている場合は間違いになかなか気づかないもんだよ．

A君　：はい，気をつけます．

● マクロ定義時の問題

N先輩：それじゃ，もう一つマクロに関する問題を出そう．リスト 56の定義で，

```
power1(x)
```

のように呼び出したら，どう展開されるかわかるよね．

A君　：ばかにしないでくださいよ．

```
((x)*(x))
```

って展開されますよ!!

N先輩：それじゃ，

```
power2(x)
```

のように呼び出したら，どう展開されるかわかるかな．

A君　：power1と同じで，

```
((x)*(x))
```

って展開されるんじゃないんですか!!

N先輩：それがね，

```
(x)  ((x)*(x))(x)
```

って展開されるんだ．

A君　：どうしてです？

N先輩：マクロの定義時にはマクロ名の次のスペースは意味をもつことになって，マクロ名の次に“(”がきてないと引き数なしのマクロ定義になってしまうんだ．

A君　：だから，power2(x)だとpower2の部分が置換されることになるから，

```
(x)  ((x)*(x))(x)
```

って展開されるんですね．でも，無意識にやってしまいそうな気がしますね．

N先輩：たいていはコンパイルエラーになるから，気づくと思うけどね．それじゃ，マクロ呼び出しのときに，

```
power1(x)
```

とした場合と，

```
power1 (x)
```

とした場合じゃ，どう違うかわかるかい．

A君　：えっと…

N先輩：そう悩まなくても，マクロ呼び出しのときにはまったく同じように展開されるんだよ．

A君　：そうですか．それを聞いて安心しました．

● マクロと型定義

N先輩：これも初心者はよく間違うんだけど，

```
#define PCHAR1  char*
typedef char*   PCHAR2;
```

で，

```
PCHAR1 pa, pb;
PCHAR2 pa, pb;
```

とした場合の違いがわかるかい．

A君　：どちらも，pa,pbがcharへのポインタになるんじゃないですか？

N先輩：それが，ちょっと違うんだ．

```
PCHAR1 pa, pb;
```

だと，

```
char* pa, pb;
```

というように展開されるから，paはcharへのポインタで，pbはchar変数となってしまうんだ．

```
PCHAR2 pa, pb;
```

の場合はどちらもcharへのポインタになるんだけどね．

A君　：そうなんですか．だったら，型定義をするときはマクロよりもtypedefを使ったほうが良いということですね．

N先輩：そういうことだね．

● TRUE，FALSE

N先輩：A君はTRUE，FALSEっていうマクロを定義しているかい．

A君　：真と偽のやつですよね．

```
#define TRUE 1
#define FALSE 0
```

って具合に定義してますよ．

N先輩：それじゃ，リスト 57はどこがまずいと思う？

A君　：とくに問題ないんじゃないですか．

N先輩：flagにTRUE，FALSEがかならず代入されれば問題ないんだけど，別の人がプログラムを改造したときに，

```
flag = 0xff;
```

なんてされると誤動作するようになるよね．

A君　：そうですね．でも，それは，

```
flag = TRUE;
```

ってしない人が悪いんじゃないですか．

N先輩：確かにそうなんだけど，Cのif文では“0”で偽，“0”以外で真っていう決まりがあるから，

```
if (flag == TRUE)
```

〔リスト 55〕マクロのトラブル（その2）

```
#define VAL1        1;
#define VAL2        2;

int flag, xval;

void func(void)
{
    if (flag)
        xval = VAL1;    /* ; がふたつになって エラー となる */
    else
        xval = VAL2;
}
```

〔リスト 56〕マクロ定義時の空白

```
#define power1(x)   ((x)*(x))
#define power2 (x)  ((x)*(x))
```

よりも，

    if (flag)

と書くようにしたほうが良いんだよ．そうすれば，

    flag = 0xff;

と，

    flag = TRUE;

のどちらでも良くなるだろ．

A君　：それじゃ，

    if (flag != FALSE)

って書くのはどうですか．

N先輩：これはプログラムの読みやすさの問題で，

    if (flag != FALSE)

だとflagが偽でないと読んでしまうけど，

    if (flag)

だったら，flagが真だったらと読むだろ．

A君　：そうですね．

N先輩：どちらが，読みやすいかを考えれば答が出るだろ．

A君　：いわれてみれば，if(flag)のほうが読みやすいですね．それじゃ，

    if (flag == FALSE)

よりも，

    if (!flag)

のほうが良いんですかね．

N先輩：FALSEを0以外に定義することはないと思うから，

    if (flag == FALSE)

でもかまわないと思うけど，僕個人としては，

    if (!flag)

のほうがCらしい気がするね．

## 移植性編

―― charの符号拡張と零拡張，ビットフィールドの並び，little endianとbig endian，86系のポインタサイズの違い

### ● charの符号拡張と零拡張

N先輩：ANSI CはK&R Cに比べるとわりと細かいところまで規定されてるけど，処理系依存の部分も残されているんだ．たとえば，charは標準的には符号付きだけど，処理系によってはコンパイルスイッチで符号なしに変更できたりするんだ．

A君　：そうですね．

N先輩：それじゃ，今度も簡単な問題だけどリスト58はどういう表示になるかわかるかい．

▶ 文字のはずが符号拡張されると…

A君　：どちらもffって表示されるんじゃないんですか．

N先輩：実はね，charが符号付きだと，

```
printf("%02x¥n", cval);    /* ffff */
printf("%02x¥n", (unsigned)cval);
                            /* ffff */
```

〔リスト57〕 TRUEとの比較

```
int flag;

void func(void)
{
    if (flag == TRUE) {
        /* ... */
    }
}
```

〔リスト58〕 charの符号拡張

```
#include <stdio.h>

void func(void)
{
    char cval = 0xff;

    printf("%02x¥n", cval);
    printf("%02x¥n", (unsigned)cval);
}
```

```
printf("%02x¥n", cval & 0xff);
                              /* ff   */
printf("%02x¥n", (unsigned char)
                    cval);   /* ff   */
```

という表示になるんだ．

A君　：えっ～．

N先輩：最初の二つはcvalが符号拡張されるからffffになるんだ．ffと表示させたかったら，後の二つのようにするか，charを符号なしでコンパイルしないといけないんだ．実際にコンパイルして実行してごらん．

A君　：…本当だ．

▶ やっかい者"符号付きchar"に注意

N先輩：符号付きのchar(signed char)は何かと問題を起こしてくれるもんなんだよ．誤動作の原因は「Cでは基本的にcharはintに符号拡張される」という大原則から生じるんだ(ANSI Cだとプロトタイプ宣言などの採用でかならずしもintに拡張されない場合もあるんだけど)．リスト58はchar変数cvalの内容を16進2桁で表示させようとするプログラムだけど，

    ffff

と4桁で表示されてしまうんだ．僕もCでプログラムを書き始めたころにこのトラブルに出会って，原因がわからないから，コンパイラ(ライブラリ)のバグだと思い違いしたことがあるんだ．charが符号付きの場合，char変数cvalを零拡張するには，

    (unsigned char)cval

とキャストする必要があるんだ．上位バイトを零拡張するには，

    cval & 0xff

としても同じ結果になるけどね．キャストを，

    (unsigned)cval

のようにすると，

    char → int → unsigned

という変換が行われるから，上位バイトが符号拡張されてしまうんだ．

A君　：へえ～，そうなんですか．

▶ 漢字を使ったときに動作が不安定になる

N先輩：charの符号拡張は日本語文字列を処理するときには中級者以上の人でも間違えてしまうことが少なくないんだ．僕もユーティリティを作成するときなどに間違えたことがあるよ．たとえば，cvalをchar変数とすると，

```
isalpha(cval)
```

という何の変哲もない例でも簡単に誤動作してしまうんだよ．

A君　：どうしてなんですか．

N先輩：さっきの例だとisalphaは0～0xffおよび－1（EOF）の引き数しか対応できないんだけど，cvalが日本語文字を処理するときにはintに符号拡張されると，対応範囲を超えてしまいうんだ．たとえば，

```
cval = 0xb1
```

だとisalpha(cval)は，

```
isalpha(0xffb1)
```

を処理することになるんだ．これだとisalphaが参照するテーブルの範囲を超えてしまうから，動作させるたびに動いたり動かなかったりという不安定な動作になるんだ．カタカナを，

```
'ｱ'
```

のように表現した場合でも符号拡張されてしまうから注意が必要だよ．

▶ 対処法はchar変数を符号なしと明示すること

N先輩：これに対処するにはさっき言ったように，

```
isalpha((unsigned char)cval)
```

とすれば良いんだけど，日本語文字列を処理する場合，コンパイルスイッチでcharを符号なしにするというのがもっとも簡単なんだ［ただし，charを符号付きとみなしているプログラムの場合は変数をunsigned charとして宣言するか(unsigned char)とキャストを付けなければならない］．ちなみにこのあたりのことを考慮しているのか16ビット版のMS-C/C++，VC++やTurbo C/C++，BC++の<jctype.h>を覗いてみるとiskanji，iskanaなどの日本語文字を扱うマクロが，

```
(unsigned char)
```

でキャストしてあるから一度眺めてみるといいよ．32ビット版のC++ BuilderやVC++のmbctype.hでも_ismbbleadなどの日本語文字を扱うマクロが，

```
(unsigned char)
```

でキャストしてあるよ．もっとも，<ctype.h>内のマクロにはキャストはないんだけどね．

A君　：はい，わかりました．

● ビットフィールドの並び

N先輩：A君はビットフィールドは知ってるかい．

A君　：はい．

N先輩：ANSI Cではビットフィールドのメンバの型は，

```
int
unsigned int
signed int
```

が認められているんだ．古いC（K&R C）だと，基本的にunsigned intだったんだけど，unsigned char，char，unsigned long，longなども可能な処理系があるんだ．intが16ビットのコンパイラに移植する場合，ビットフィールドにunsigned longを使用していると宣言の書き直しだけで対応できない可能性があって，intのサイズが違うとビットフィールドの境界合わせがずれる可能性もあるんだ．

A君　：たいへんそうですね．

N先輩：でも，ビットフィールドでもっとも問題になるのはビットの割り付け順序なんだ．ビットフィールドを上位ビットから割り付けるか，下位ビットから割り付けるかはANSI Cでも処理系依存になってるんだ．クロスコンパイラだとコンパイルスイッチで切り換えられるものもあるけど，切り換えられないコンパイラだと移行がたいへんなんだよ．ビットフィールドはメモリマップドI/OのシステムでのI/O操作なんかに使うとプログラムが見やすくなって便利なんだけど，一般に生成コードはあまりよくならないから，&や|で操作するプログラムを組んでおくほうが移植性もよくなるものなんだよ（リスト59，リスト60）．

A君　：へえー，そうなんですか．

● little endianとbig endian

N先輩：little endianとbig endianというのを知ってるかい．

A君　：いいえ．

N先輩：Windows系のソフトウェアしか開発しないという人ならともかく，われわれのように組み込み系の開発で86系と68K系の両方を使う場合は，little endianとbig endianは頭に入れておかないといけないよ．メモリ内に，

```
0x12, 0x34, 0x56, 0x78
```

という順に並んでいた場合，68K系だと下位番地が上位バイトになるから（big endian），

```
0x12345678L
```

となるんだけど，86系では下位番地が下位バイトになるから（little endian），

```
0x78563412L
```

となるんだ．どちらの方式にも利点と欠点があるから，TRONチップのように切り換えができれば便利なんだ

〔リスト59〕ビットフィールド

```
struct {
    unsigned bit0 : 1;
    unsigned bit1 : 1;
    unsigned bit2 : 1;
    unsigned bit3 : 1;
    unsigned bit4 : 1;
    unsigned bit5 : 1;
    unsigned bit6 : 1;
    unsigned bit7 : 1;
    unsigned bit8 : 1;
    unsigned bit9 : 1;
    unsigned bita : 1;
    unsigned bitb : 1;
    unsigned bitc : 1;
    unsigned bitd : 1;
    unsigned bite : 1;
    unsigned bitf : 1;
} s;

void func(void)
{
    if (s.bit0) {
        /* ... */;
    } else if (s.bitf) {
        /* ... */;
    }
}
```

〔リスト60〕&によるビットテスト

```
enum bitvals {BIT0 = 0x0001,
              BIT1 = 0x0002,
              BIT2 = 0x0004,
              BIT3 = 0x0008,
              BIT4 = 0x0010,
              BIT5 = 0x0020,
              BIT6 = 0x0040,
              BIT7 = 0x0080,
              BIT8 = 0x0100,
              BIT9 = 0x0200,
              BITa = 0x0400,
              BITb = 0x0800,
              BITc = 0x1000,
              BITd = 0x2000,
              BITe = 0x4000,
              BITf = (int)0x8000};

enum bitvals s;

void func(void)
{
    if (s & BIT0) {
        /* ... */;
    } else if (s & BITf) {
        /* ... */;
    }
}
```

〔リスト62〕ポインタサイズの違い

```
#include <stddef.h>

void subcode(void (*)());
void subdata(void *);

void func(void)          /* NULL ポインタサイズ の違い      */
{                        /* スモール  ミディアム コンパクト ラージ */
    subcode(NULL);       /* 2 バイト   4 バイト   2 バイト   4 バイト */
    subdata(NULL);       /* 2 バイト   2 バイト   4 バイト   4 バイト */
}
```

〔リスト61〕endianの問題点

```
#include <stdio.h>

void func(void)
{
    int ch = 0;

    scanf("%c", &ch);
    /* ... */;
}
```

〔リスト63〕浮動小数点の比較

```
#define FLOATCONST (1.0 / 3.0)

void func(void)
{
    float floatvar;

    floatvar = FLOATCONST;
    /* ... */;
    if (floatvar == FLOATCONST) {
        /* ... */;
    }
}
```

けど．Cでプログラムを組んでるかぎりはこの違いが表に出てくることは少ないんだけど，デバッガのようにメモリ内を直接操作するときには注意しないといけないよ．

A君　：はい，わかりました．

N先輩：ところで，リスト61はどこが間違ってると思う？

A君　：えっと一…．scanfではint変数chの先頭のバイトにcharデータがセットされるから，little endianだとうまくいきますけど，big endianだとおかしくなっちゃいますね．

N先輩：そういうことだね．リスト61のような場合にはchar ch;としなくちゃいけないんだ．

A君　：そうですね．

### ● 86系のポインタサイズの違い

N先輩：他のCPUとは違って，16ビット版の86系にはメモリモデルというのがあるんだけど知ってるよね．

A君　：スモールモデル，ラージモデルというやつですよね．

N先輩：メモリモデルは16ビット版の86系のCPUを効率よく使用するためには必要なものなんだけど，データ領域（変数）へのポインタとコード領域（関数）へのポインタのサイズが異なるメモリモデルだとポインタのサイズに注意する必要があるんだ．たとえば，リスト62だとコンパクトモデルやミディアムモデルでは同じNULLでもバイト数が異なるんだ．

A君　：ややこしいですね．

N先輩：リスト62のようにまともなプロトタイプ宣言があれば適切な変換が行われるから，問題となることはないんだけど，可変引き数の関数などでプロトタイプ宣言が働かない場合にはキャストで明示的に指示しないと正常なコードが生成されないことがあるんだ．86系の初期時代の話だけど，K&R Cレベルのコンパイラがコンパクトモデルやミディアムモデルをサポートしていなかったのはこのあたりに原因の一端があるんだ．

A君　：僕はスモールモデルとラージモデルしか使ったことがないですから，今後もミディアムモデルやコンパクトモデルは使わないようにします．

N先輩：別にそこまでしなくても良いよ．スモールモデルで収まらない場合はラージモデルにしたほうが変なトラブルに悩まなくて済むかもしれないね．

A君　：はい．

## 浮動小数点編

――浮動小数点の比較（一致，不一致），浮動小数点の比較（その2）

### ● 浮動小数点の比較（一致，不一致）

N先輩：A君はリスト63でどこが間違っているかわかるかい．

A君　：代入した値と比較しているだけで，どこか間違っているんですか．

N先輩：floatvarとFLOATCONSTの比較をfloatで行うようなコードを生成するコンパイラだとうまくいくんだけど，doubleやlong doubleで行うコードを生

成するコンパイラだとうまくいかないんだ．

A君　：どうしてです？

N先輩：比較を doubleで行うコードが生成されるとすると floatvarは一度，floatに丸めた値を doubleに拡張した値になって，FLOATCONSTは floatに丸めないで，直接 doubleに丸めた値になるから，一致しないことがあるんだ．

A君　：FLOATCONSTは 1.0/3.0で循環小数だから，floatに丸めた値と doubleに丸めた値が違ってくるということですね．

N先輩：そういうことだね．一度，floatに丸めた値を doubleに拡張しても，doubleの精度にはならないからね．

### ● 浮動小数点の比較（その2）

N先輩：それじゃ，リスト 64が間違っているのはわかるよね．

A君　：浮動小数点は丸められた値だから，一致，不一致の比較はだめですよね．

〔リスト 64〕浮動小数点の比較（間違い）

```
double da, db;

void func(void)
{
    if(da == db)
        /* ... */;
}
```

〔リスト 65〕浮動小数点の比較

```
#include <math.h>

double epsilon = 1.0E-07;
double da, db;

void func(void)
{
    if (fabs(da - db) < epsilon * fabs(da))
        /* ... */;
}
```

〔リスト 66〕浮動小数点の比較

```
#include <math.h>
#include <float.h>

double da, db;

void func(void)
{
    if (fabs(da - db) < DBL_EPSILON)
        /* ... */;
}
```

〔リスト 67〕EOF

```
#include <stdio.h>

void func(void)
{
    char ch;

    while ((ch = getchar()) != EOF) {
        /* ... */;
    }
}
```

N先輩：だったらどうすれば良いかわかるかな．

A君　：えっーと，引き算して基準値以下だったら，一致で，大きかったら不一致って感じですかね．

N先輩：そうだね．基準値をどうやって決めるかってのが難しいけどね．たとえば，リスト 65のように比較する変数を基準にして決めるか…．

A君　：でも，リスト 65だと daが 0だとだめですよね．

N先輩：そうだね．アプリケーションによって変えないといけないだろうね．VC++，BC++，C++ Builderなんかだと float.hに DBL_EPSILONが定義してあるから，リスト 66のようにするって方法もあると思うよ．

## その他
――― EOFの定義，現在位置にシーク

### ● EOFの定義

N先輩：リスト 67は単純なミスだけど，どこが間違ってると思う？

A君　：どこって，おかしいとしたら，whileのとこですよね．

N先輩：getchar()関数の返り値は intだから，char変数 chに代入すると誤動作するんだ．getchar()関数は 0～0xffの範囲の正常な返り値があって，ファイルの終わりか，またはエラーの場合に EOFを返すんだ．だから，

```
char ch;
```

を，

```
int ch;
```

にしないといけないんだ．

A君　：あっ，そうですね．

N先輩：EOFは－1と定義されてるから，たまたま，ファイルがテキストファイルだったりして，0xffが含まれていないと正常に動作するんだけどね．

### ● 現在位置にシーク〔fseek(fp, 0L, SEEK_CUR)〕

N先輩：fseek関数は知ってるかい．

A君　：はい．

N先輩：それじゃ，fseek(fp, 0L, SEEK_CUR)の意味はわかるかな．

A君　：現在位置にシークすることですけど，なんの意味があるんですかね．

N先輩：フォーマットの決まったファイルをアクセスする場合にはシークしながらアクセスする必要があるんだ．そのため，ヘッダファイル <stdio.h>にはファイルシーク用として，

```
#define SEEK_SET 0
        /* ファイルの先頭からの絶対位置 */
#define SEEK_CUR 1
        /* 現在の位置からの相対位置     */
#define SEEK_END 2
```

〔リスト68〕
fseek( fp, 0L, SEEK_CUR);

```
#include <stdio.h>

char sopt[] = "abcdef";
extern char b2eopts[];

int func(char pexe[])
{
    FILE *fp;
    int ch;
    char *popts;

    if ((fp = fopen(pexe, "r+b")) == NULL) {
        fprintf(stderr, "ファイル %s が読み込めません.¥n", pexe);
        return 1;
    }
    popts = b2eopts;
    while ((ch = getc(fp)) != EOF) {
        if ((char)ch == *popts) {   /* 識別文字列の検索      */
            if (!*popts++) {
                fseek(fp, 0L, SEEK_CUR);    /* 現在位置に シーク    */
                if (fwrite(sopt, sizeof(sopt), 1, fp) == 1) {
                    fclose(fp);
                    return 0;
                } else {
                    fprintf(stderr, "ファイル %s の更新に失敗しました.¥n", pexe);
                    fclose(fp);
                    return 1;
                }
            }
        } else
            popts = b2eopts;
    }
    fprintf(stderr, "ファイル %s の更新位置が見つかりません.¥n", pexe);
    fclose(fp);
    return 1;
}
```

/* ファイルの最後からの相対位置 */

というマクロが定義されているんだ．でも，ファイルポインタ(`FILE *`)によるアクセスだとバッファが介在するから，ファイルの更新で思うようにいかないことがあるんだ．そのときのおまじないが現在位置にシークする，

`fseek(fp, 0L, SEEK_CUR)`

なんだよ．たとえば，リスト68はファイル中の識別文字列を検索した後にファイルを更新するプログラムを抜き出したものなんだけど，現在位置にシークする一見無意味に見える，

`fseek(fp, 0L, SEEK_CUR);`

がないとファイルをうまく更新できないんだ．A君もいずれこの現象に悩むことがあると思うから，頭の片隅に置いておけば役に立つと思うよ．

A君　：はい，わかりました．

## おわりに

本章ではCプログラミングで間違いやすいコーディング例のいくつかを紹介しました．Windows上のプログラミングではC言語が使われることは少なくなりましたが，組み込み分野ではまだ多く使われています．本章の例を役立ててください．

なかしま・のぶゆき　(株)Unix

**第2章**

VC++1.5とVC++5.0でコンパイル結果を比べてみる

# コーディングの違いと最適化例

中島 信行



Cではポインタと配列は，コーディングする上では同じように記述できる場合があります．Cらしさではポインタ風のコーディングということになりますが，読みやすさの点では配列風のコーディングが良いこともあります．本章ではコーディングの違いによって，最適化がどのように影響を受けるのか，いくつか例をあげてみていきたいと思います．（筆者）

A君　：先輩，ちょっと質問があるんですけど．

N先輩：なんだい．

A君　：Cだとポインタと配列は，変数と定数という違いはありますけど，コーディング上は同じように記述できる場合がありますよね．

N先輩：そうだね．

A君　：そういうときは，どっちが良いんですかね．

N先輩：最近のコンパイラの最適化はかなり良くなってきたから，プログラムが読みやすくなるように書けば良いんだけど，コーディングの違いが最適化にどのように影響するかをいくつか例を上げてVC++ 1.5（16ビット）とVC++ 5.0（32ビット）で調べてみようか．

A君　：そうですね．そのほうが参考になりそうですね．

## ポインタと配列風コーディングの違いと最適化

N先輩：リスト1のmemcpy1関数とmemcpy2関数は配列風な書き方とポインタの書き方の違いだけで同一のコードだよね．でも，VC++ 1.5だとリスト2を見てみるとlodsb命令を使うコードを生成しているmemcpy2関数のほうがループ内の処理が短くなってるんだ．memcpy3関数はmemcpy2関数をさらに，Cらしく記述したものだけど，こちらはlodsb命令を使ったコードを生成しているものの，後置デクリメント（size--）の部分のコードが悪いから，ループ内のコードが長くなってるね（後置デクリメントに関しては後述）．

A君　：そうですね．

N先輩：VC++ 5.0の場合をリスト3（最大限の最適化：/c /Ox）に示すよ．VC++ 5.0だとlodsb，stosb命令ともに使われてないね．関数全体のバイト数だとmemcpy2関数がもっとも短いコードになってるんだけど，ループ内のクロック数に着目してみると，

memcpy1 ＞ memcpy2 ＝ memcpy3

の順で配列風のコーディングがもっとも高速なコード

〔リスト1〕ポインタと配列風コーディング

```
void memcpy1(char pd[], const char ps[], unsigned size)
{
    unsigned n;

    for (n = 0; n < size; n++)
        pd[n] = ps[n];
}

void memcpy2(char *pd, const char *ps, unsigned size)
{
    unsigned n;

    for (n = 0; n < size; n++)
        *pd++ = *ps++;
}

void memcpy3(char *pd, const char *ps, unsigned size)
{
    while (size--)
        *pd++ = *ps++;
}
```

〔リスト2〕リスト1のアセンブリリスト
（VC++ 1.5，コマンドラインオプション/c /Ox）

```
;|*** void memcpy1(char pd[], const char ps[], unsigned
size)
;|*** {
    ..................
;   pd = 4
;   ps = 6
;   size = 8
;   n = -4
    *** 000008  8b 56 08       mov dx,WORD PTR [bp+8]   ;size
;|***     unsigned n;
;|***
;|***     for (n = 0; n < size; n++)
; ライン 5
    *** 00000b  0b d2          or  dx,dx
    *** 00000d  74 0f          je  $EX177
    *** 00000f  8b 7e 06       mov di,WORD PTR [bp+6]   ;ps
    *** 000012  8b 76 04       mov si,WORD PTR [bp+4]   ;pd
                          $F179:
;|***          pd[n] = ps[n];
; ライン 6
    *** 000015  8a 05          mov al,BYTE PTR [di]
```

〔リスト 2〕リスト 1 のアセンブリリスト(VC++ 1.5，コマンドラインオプション/c /Ox)(つづき)

```
    *** 000017  47              inc di
    *** 000018  88 04           mov BYTE PTR [si],al
    *** 00001a  46              inc si
    *** 00001b  4a              dec dx
    *** 00001c  75 f7           jne $F179
;|*** }
; ライン 7
                $EX177:
    ..................
;|*** void memcpy2(char *pd, const char *ps, unsigned size)
;|*** {
    ..................
;   pd = 4
;   ps = 6
;   size = 8
;   n = -2
    *** 00002c  8b 5e 08        mov bx,WORD PTR [bp+8]  ;size
;|***     unsigned n;
;|***
;|***     for (n = 0; n < size; n++)
; ライン 13
    *** 00002f  0b db           or  bx,bx
    *** 000031  74 0f           je  $EX189
    *** 000033  8b d3           mov dx,bx
    *** 000035  8b 76 06        mov si,WORD PTR [bp+6]  ;ps
    *** 000038  8b 7e 04        mov di,WORD PTR [bp+4]  ;pd
                $F191:
;|***         *pd++ = *ps++;
; ライン 14
    *** 00003b  ac              lodsb
    *** 00003c  88 05           mov BYTE PTR [di],al
    *** 00003e  47              inc di
    *** 00003f  4a              dec dx
    *** 000040  75 f9           jne $F191
;|*** }
; ライン 15
                $EX189:
    ..................
;|*** void memcpy3(char *pd, const char *ps, unsigned size)
;|*** {
    ..................
;   pd = 4
;   ps = 6
;   size = 8
    *** 00004d  8b 76 06        mov si,WORD PTR [bp+6]  ;ps
    *** 000050  8b 7e 04        mov di,WORD PTR [bp+4]  ;pd
    *** 000053  8b 56 08        mov dx,WORD PTR [bp+8]  ;size
;|***     while (size--)
; ライン 19
    *** 000056  eb 04           jmp SHORT $L211
                $FC203:
;|***         *pd++ = *ps++;
; ライン 20
    *** 000058  ac              lodsb
    *** 000059  88 05           mov BYTE PTR [di],al
    *** 00005b  47              inc di
                $L211:
    *** 00005c  8b c2           mov ax,dx
    *** 00005e  4a              dec dx
    *** 00005f  0b c0           or  ax,ax
    *** 000061  75 f5           jne $FC203
;|*** }
```

〔リスト 3〕リスト 1 のアセンブリリスト(VC++ 5.0，コマンドラインオプション/c /Ox)

```
_pd$ = 8
_ps$ = 12
_size$ = 16
_memcpy1 PROC NEAR

; 2    : {

  00000 56          push    esi

; 3    :     unsigned n;
; 4    :
; 5    :     for (n = 0; n < size; n++)

  00001 8b 74 24 10  mov     esi, DWORD PTR _size$[esp]
  00005 85 f6        test    esi, esi
  00007 76 13        jbe     SHORT $L89
  00009 8b 44 24 08  mov     eax, DWORD PTR _pd$[esp]
  0000d 8b 4c 24 0c  mov     ecx, DWORD PTR _ps$[esp]
  00011 2b c8        sub     ecx, eax
$L87:

; 6    :         pd[n] = ps[n];

  00013 8a 14 01     mov     dl, BYTE PTR [ecx+eax]
  00016 88 10        mov     BYTE PTR [eax], dl
  00018 40           inc     eax
  00019 4e           dec     esi
  0001a 75 f7        jne     SHORT $L87
$L89:
    ..................
_pd$ = 8
_ps$ = 12
_size$ = 16
_memcpy2 PROC NEAR

; 10   : {

  00020 56           push    esi

; 11   :     unsigned n;
; 12   :
; 13   :     for (n = 0; n < size; n++)

  00021 8b 74 24 10  mov     esi, DWORD PTR _size$[esp]
  00025 85 f6        test    esi, esi
  00027 76 11        jbe     SHORT $L98
  00029 8b 4c 24 0c  mov     ecx, DWORD PTR _ps$[esp]
  0002d 8b 44 24 08  mov     eax, DWORD PTR _pd$[esp]
$L96:

; 14   :         *pd++ = *ps++;

  00031 8a 11        mov     dl, BYTE PTR [ecx]
  00033 88 10        mov     BYTE PTR [eax], dl
  00035 40           inc     eax
  00036 41           inc     ecx
  00037 4e           dec     esi
  00038 75 f7        jne     SHORT $L96
$L98:
    ..................
_pd$ = 8
_ps$ = 12
_size$ = 16
_memcpy3 PROC NEAR

; 19   :     while (size--)

  00040 8b 44 24 0c  mov     eax, DWORD PTR _size$[esp-4]
  00044 8b c8        mov     ecx, eax
  00046 48           dec     eax
  00047 85 c9        test    ecx, ecx
  00049 74 16        je  SHORT $L106
  0004b 8b 4c 24 08  mov     ecx, DWORD PTR _ps$[esp-4]
  0004f 56           push    esi
  00050 8d 70 01     lea     esi, DWORD PTR [eax+1]
  00053 8b 44 24 08  mov     eax, DWORD PTR _pd$[esp]
$L105:

; 20   :         *pd++ = *ps++;

  00057 8a 11        mov     dl, BYTE PTR [ecx]
  00059 88 10        mov     BYTE PTR [eax], dl
  0005b 40           inc     eax
  0005c 41           inc     ecx
  0005d 4e           dec     esi
  0005e 75 f7        jne     SHORT $L105

; 18   : {

  00060 5e           pop     esi
$L106:
```

〔リスト 4〕while ループと for ループ

```
void zrclr1(char *pd, unsigned size)
{
    while (size--)
        *pd++ = 0;
}

void zrclr2(char *pd, unsigned size)
{
    for (; size--;)
        *pd++ = 0;
}

void zrclr3(char *pd, unsigned size)
{
    for (; size; --size)
        *pd++ = 0;
          /* /O1でループがrep stosに置き変わる */
}

void zrclr4(char *pd, unsigned size)
{
    unsigned n;

    for (n = 0; n < size; n++)
        *pd++ = 0;
          /* /O1でループがrep stosに置き変わる */
}

void zrclr5(char *pd, unsigned size)
{
    unsigned n;

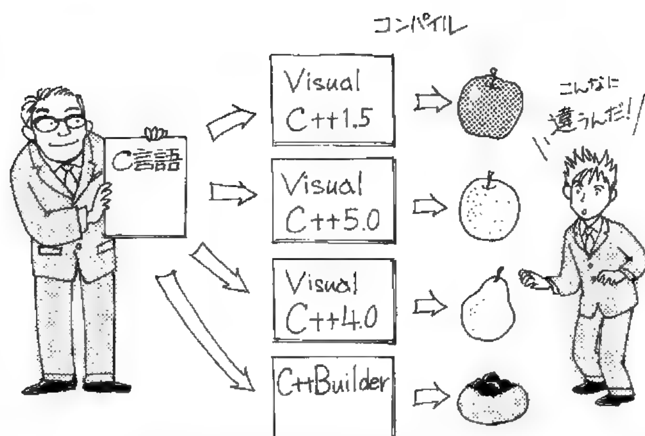
    for (n = 0; n < size; n++)
        pd[n] = 0;
          /* /O1でループがrep stosに置き変わる */
}
```

〔リスト 5〕リスト 4のアセンブリリスト( VC++ 1.5，コマンドラインオプション/c /Ox)

```
;|*** void zrclr1(char *pd, unsigned size)
;|*** {
; ライン 2
    *** 000000  55          push    bp
    *** 000001  8b ec       mov bp,sp
;   pd = 4
;   size = 6
    *** 000003  8b 5e 04    mov bx,WORD PTR [bp+4]  ;pd
    *** 000006  8b 56 06    mov dx,WORD PTR [bp+6]  ;size
;|***     while (size--)
; ライン 3
    *** 000009  eb 05       jmp SHORT $L228
    *** 00000b  90          nop
                    $FC177:
;|***         *pd++ = 0;
; ライン 4
    *** 00000c  c6 07 00    mov BYTE PTR [bx],0
    *** 00000f  43          inc bx
                    $L228:
    *** 000010  8b c2       mov ax,dx
    *** 000012  4a          dec dx
    *** 000013  0b c0       or  ax,ax
    *** 000015  75 f5       jne $FC177
;|*** }
    ...................
;|*** void zrclr2(char *pd, unsigned size)
;|*** {
;|***     for (; size--;)
;|***         *pd++ = 0;
    ...................   zrclr1と同一の生成コード

;|*** void zrclr3(char *pd, unsigned size)
;|*** {
; ライン 14
    *** 000038  55          push    bp
    *** 000039  8b ec       mov bp,sp
    *** 00003b  57          push    di
;   pd = 4
;   size = 6
    *** 00003c  8b 56 06    mov dx,WORD PTR [bp+6]  ;size
;|***     for (; size; --size)
; ライン 15
    *** 00003f  0b d2       or  dx,dx
    *** 000041  74 12       je  $L221
    *** 000043  33 c0       xor ax,ax
    *** 000045  8b 5e 04    mov bx,WORD PTR [bp+4]  ;pd
    *** 000048  8b ca       mov cx,dx
    *** 00004a  8b fb       mov di,bx
    *** 00004c  1e          push    ds
    *** 00004d  07          pop es
    *** 00004e  d1 e9       shr cx,1
    *** 000050  f3          rep
    *** 000051  ab          stosw
    *** 000052  73 01       jae $L221
    *** 000054  aa          stosb
                    $L221:
;|***         *pd++ = 0;    /* /O1でループがrep stosに置き変わる */
;|*** }
    ...................
;|*** void zrclr4(char *pd, unsigned size)
;|*** {
;|***     for (n = 0; n < size; n++)
;|***         *pd++ = 0;    /* /O1でループがrep stosに置き変わる */
    ...................   zrclr3と同一の生成コード

;|*** void zrclr5(char *pd, unsigned size)
;|*** {
;|***     for (n = 0; n < size; n++)
;|***         pd[n] = 0;    /* /O1でループがrep stosに置き変わる */
    ...................   zrclr3と同一の生成コード
```




になってるんだ．ちなみに，VC++ 4.0だと memcpy3 のループ内は8命令もあって，もっとも効率が悪かったんだけど，後置デクリメントの生成コードも着実に改善されてきているようだね( C++ Builder の-O2オプションでは func1，func2 関数のループ内が7命令，func3 関数は8命令だった)．

## while ループと for ループの違いと最適化

N先輩：while ループと for ループは親戚みたいなものだけど，生成コードに違いがあるかどうか調べてみよう．リスト 4とリスト 5を見てごらん．VC++ 1.5だと

〔リスト6〕リスト4のアセンブリリスト（VC++ 5.0, コマンドラインオプション/c /Ox）

```
_pd$ = 8
_size$ = 12
_zrclr1 PROC NEAR

; 3     :       while (size--)

  00000 8b 44 24 08  mov     eax, DWORD PTR _size$[esp-4]
  00004 8b c8        mov     ecx, eax
  00006 48           dec     eax
  00007 85 c9        test    ecx, ecx
  00009 74 19        je  SHORT $L86
  0000b 8d 48 01     lea     ecx, DWORD PTR [eax+1]
  0000e 57           push    edi
  0000f 8b 7c 24 08  mov     edi, DWORD PTR _pd$[esp]
  00013 8b d1        mov     edx, ecx
  00015 33 c0        xor     eax, eax
  00017 c1 e9 02     shr     ecx, 2
  0001a f3 ab        rep stosd
  0001c 8b ca        mov     ecx, edx
  0001e 83 e1 03     and     ecx, 3
  00021 f3 aa        rep stosb
  00023 5f           pop     edi
$L86:

; 4     :           *pd++ = 0;
; 5     : }
    ...................
_zrclr2 PROC NEAR

; 9     :       for (; size--;)
; 10    :           *pd++ = 0;
    ...................   zrclr1と同一の生成コード
```

```
_pd$ = 8
_size$ = 12
_zrclr3 PROC NEAR

; 15    :       for (; size; --size)

  00060 8b 4c 24 08  mov     ecx, DWORD PTR _size$[esp-4]
  00064 85 c9        test    ecx, ecx
  00066 74 16        je  SHORT $L99
  00068 8b d1        mov     edx, ecx
  0006a 57           push    edi
  0006b 8b 7c 24 08  mov     edi, DWORD PTR _pd$[esp]
  0006f 33 c0        xor     eax, eax
  00071 c1 e9 02     shr     ecx, 2
  00074 f3 ab        rep stosd
  00076 8b ca        mov     ecx, edx
  00078 83 e1 03     and     ecx, 3
  0007b f3 aa        rep stosb
  0007d 5f           pop     edi
$L99:

; 16    :           *pd++ = 0; /* /O1でループがrep stosに置き変わる */
; 17    : }
    ...................
_zrclr4 PROC NEAR
; 23    :       for (n = 0; n < size; n++)
; 24    :           *pd++ = 0; /* /O1でループがrep stosに置き変わる */
    ...................   zrclr3と同一の生成コード
_zrclr5 PROC NEAR
; 31    :       for (n = 0; n < size; n++)
; 32    :           pd[n] = 0; /* /O1でループがrep stosに置き変わる */
    ...................   zrclr3と同一の生成コード
```

〔リスト7〕+と|

```
#define MK_FP1(s, o)    (void far *)(((unsigned long)(s) << 16) | (unsigned)(o))
#define MK_FP2(s, o)    (void far *)(((unsigned long)(s) << 16) + (unsigned)(o))

unsigned seg, off;
void far *p1, far *p2;

void func1(void)
{
    p1 = MK_FP1(seg, off);
}

void func2(void)
{
    p2 = MK_FP2(seg, off);
}
```

zrclr1関数のwhileループを単純にforループにしただけではさすがに同じコードが生成されているね．でも，zrclr3，zrclr4，zrclr5関数だとループがrep stos命令に置き変わっているよ．

A君　：そうですね．

N先輩：どうやら後置デクリメント（size--）は生成コードに悪影響を与えているみたいだ（ちなみに，BC++ 3.0ではzrclr5関数だけがrep stos命令に置き変わった）．

A君　：そのようですね．

N先輩：VC++ 5.0の場合をリスト6（最大限の最適化：/c /Ox）に示すよ．VC++ 4.0だとVC++ 1.5と同様にzrclr3，zrclr4，zrclr5関数だけループがrep stos命令に置き変わっていたんだけど，VC++ 5.0だと全部の関数がrep stos命令に置き変わってるよ（/Ogオプションでループがrep stosに置き変わる）．VC++ 5.0は後置デクリメントを克服したみたいだね（C++ Builderの-O2オプションではrep stos命令は使われない）．

## +と|の違いと最適化

N先輩：16ビットコード（VC++ 1.5）の場合，longの加算は下位ワードからの桁上がりを上位ワードに加算してやる必要がある関係で，ビットORよりも若干効率が悪くなる可能性があるんだ．それで，farポインタの生成（リスト7）のように+と|のどちらでもかまわないときには|を使うほうが生成コードが良くなる可能性があるんだ．

A君　：そうなんですか．

N先輩：でも，生成コード（リスト8）をみてみるとこの程度のことは判断してくれるようで，どちらも同じコードが生成されているね．

A君　：そうですね．

N先輩：VC++ 5.0の場合をリスト9（最大限の最適化：/c /Ox）に示すよ．VC++ 4.0の場合は|の，

```
mov     ecx, DWORD PTR _off
or      eax, ecx
```

〔リスト 8〕リスト 7のアセンブリリスト
（VC++ 1.5，コマンドラインオプション/c /Ox）

```
;|*** void func1(void)
;|*** {
;|***     p1 = MK_FP1(seg, off);
; ライン 9
    *** 000000  a1 00 00        mov ax,WORD PTR _seg
    *** 000003  8b 0e 00 00     mov cx,WORD PTR _off
    *** 000007  89 0e 00 00     mov WORD PTR _p1,cx
    *** 00000b  a3 02 00        mov WORD PTR _p1+2,ax
;|*** }
    ..................
;|*** void func2(void)
;|*** {
;|***     p2 = MK_FP2(seg, off);
; ライン 14
    *** 000010  a1 00 00        mov ax,WORD PTR _seg
    *** 000013  8b 0e 00 00     mov cx,WORD PTR _off
    *** 000017  89 0e 00 00     mov WORD PTR _p2,cx
    *** 00001b  a3 02 00        mov WORD PTR _p2+2,ax
;|*** }
```

〔リスト 9〕リスト 7のアセンブリリスト
（VC++ 5.0，コマンドラインオプション/c /Ox）

```
_func1  PROC NEAR

; 9    :     p1 = MK_FP1(seg, off);

  00000 a1 00 00 00 00   mov    eax, DWORD PTR _seg
  00005 8b 0d 00 00 00
    00                   mov    ecx, DWORD PTR _off
  0000b c1 e0 10         shl    eax, 16   ; 00000010H
  0000e 0b c1            or     eax, ecx
  00010 a3 00 00 00 00   mov    DWORD PTR _p1, eax
    ..................
_func2  PROC NEAR

; 14   :     p2 = MK_FP2(seg, off);

  00020 a1 00 00 00 00   mov    eax, DWORD PTR _seg
  00025 8b 0d 00 00 00
    00                   mov    ecx, DWORD PTR _off
  0002b c1 e0 10         shl    eax, 16   ; 00000010H
  0002e 03 c1            add    eax, ecx
  00030 a3 00 00 00 00   mov    DWORD PTR _p2, eax
```

〔リスト 10〕変数の前置デクリメントと後置デクリメント

```
/* size <= 0x7fff とする */
void zrclr1(char *pd, int size)
{
    while (size-- > 0)
        *pd++ = 0;
}

void zrclr2(char *pd, int size)
{
    while (--size >= 0)
        *pd++ = 0;
}
```

のコードが，

```
    or      eax, DWORD PTR _off
```

というコードになっていたんだけど，VC++ 5.0だと | と + が同様のコードになってるね．基本的にメモリアクセスは命令数が増えても，mov 命令を使って 1クロック命令で構成するようなコードが生成されるみたいだね（C++ Builder の-O2 オプションでは VC++ 4.0 と同等のコードが生成された）．

〔リスト 11〕リスト 10のアセンブリリスト
（VC++ 1.5，コマンドラインオプション/c /Ox）

```
;|*** void zrclr1(char *pd, int size)
;|*** {
; ライン 3
    *** 000000  55          push    bp
    *** 000001  8b ec           mov bp,sp
;   pd = 4
;   size = 6
    *** 000003  8b 5e 04        mov bx,WORD PTR [bp+4]   ;pd
    *** 000006  8b 56 06        mov dx,WORD PTR [bp+6]
;size
;|***     while (size-- > 0)
; ライン 4
    *** 000009  eb 05           jmp SHORT $L190
    *** 00000b  90          nop
                    $FC177:
;|***         *pd++ = 0;
; ライン 5
    *** 00000c  c6 07 00        mov BYTE PTR [bx],0
    *** 00000f  43          inc bx
                    $L190:
    *** 000010  8b c2           mov ax,dx
    *** 000012  4a          dec dx
    *** 000013  0b c0           or  ax,ax
    *** 000015  7f f5           jg  $FC177
;|*** }
    ..................
;|*** void zrclr2(char *pd, int size)
;|*** {
; ライン 9
    *** 00001c  55          push    bp
    *** 00001d  8b ec       mov bp,sp
;   pd = 4
;   size = 6
    *** 00001f  8b 5e 04    mov bx,WORD PTR [bp+4]   ;pd
    *** 000022  8b 56 06    mov dx,WORD PTR [bp+6]   ;size
;|***     while (--size >= 0)
; ライン 10
    *** 000025  eb 05       jmp SHORT $L191
    *** 000027  90          nop
                    $FC186:
;|***         *pd++ = 0;
; ライン 11
    *** 000028  c6 07 00    mov BYTE PTR [bx],0
    *** 00002b  43          inc bx
                    $L191:
    *** 00002c  4a          dec dx
    *** 00002d  79 f9       jns $FC186
;|*** }
```

## 変数の前置デクリメントと後置デクリメントの違いと最適化

N先輩：さっきいった変数の前置デクリメントと後置デクリメントを比較してみよう．リスト 10の zrclr1，zrclr2 関数は size <= 0x7fff である限りは同じ処理となるよね．でも，VC++ 1.5だと前置デクリメントのほうがデクリメントの結果をそのまま判定に使える関係で，生成コードが良くなっているね（リスト 11）．繰り返し回数が数十回とか数百回といった int の範囲で処理できるループは少なくないから，覚えておいて損はないかもしれないよ．

A君　：はい，頭に入れておきます．

N先輩：VC++ 5.0の場合をリスト 12（最大限の最適化：/c /Ox）に示すよ．VC++ 4.0だと後置デクリメントは苦手だったんだけど，VC++ 5.0だと後置デクリメントも

〔リスト12〕リスト10のアセンブリリスト(VC++ 5.0，コマンドラインオプション/c /Ox)

```
_pd$ = 8
_size$ = 12
_zrclr1 PROC NEAR

; 4    :     while (size-- > 0)

  00000 8b 44 24 08  mov     eax, DWORD PTR _size$[esp-4]
  00004 8b c8        mov     ecx, eax
  00006 48           dec     eax
  00007 85 c9        test    ecx, ecx
  00009 7e 19        jle     SHORT $L86
  0000b 8d 48 01     lea     ecx, DWORD PTR [eax+1]
  0000e 57           push    edi
  0000f 8b 7c 24 08  mov     edi, DWORD PTR _pd$[esp]
  00013 8b d1        mov     edx, ecx
  00015 33 c0        xor     eax, eax
  00017 c1 e9 02     shr     ecx, 2
  0001a f3 ab        rep stosd
  0001c 8b ca        mov     ecx, edx
  0001e 83 e1 03     and     ecx, 3
  00021 f3 aa        rep stosb
  00023 5f           pop     edi
$L86:

; 5    :         *pd++ = 0;
; 6    : }
```

```
    ...................
_pd$ = 8
_size$ = 12
_zrclr2 PROC NEAR

; 10   :     while (--size >= 0)

  00030 8b 44 24 08  mov     eax, DWORD PTR _size$[esp-4]
  00034 48           dec     eax
  00035 78 19        js  SHORT $L93
  00037 8d 48 01     lea     ecx, DWORD PTR [eax+1]
  0003a 57           push    edi
  0003b 8b 7c 24 08  mov     edi, DWORD PTR _pd$[esp]
  0003f 8b d1        mov     edx, ecx
  00041 33 c0        xor     eax, eax
  00043 c1 e9 02     shr     ecx, 2
  00046 f3 ab        rep stosd
  00048 8b ca        mov     ecx, edx
  0004a 83 e1 03     and     ecx, 3
  0004d f3 aa        rep stosb
  0004f 5f           pop     edi
$L93:

; 11   :         *pd++ = 0;
; 12   : }
```

〔リスト13〕ポインタの前置/後置インクリメント，前置/後置デクリメント

```
void *memmove1(void *dst, const void *src, unsigned n)
{
    char *pd = dst;
    const char *ps = src;

    if (dst < src) {
        for (; n; n--)
            *pd++ = *ps++;
    } else if (dst > src) {
        for (pd += n - 1, ps += n - 1; n; n--)
            *pd-- = *ps--;
    }
    return dst;
}
```

```
void *memmove2(void *dst, const void *src, unsigned n)
{
    char *pd = dst;
    const char *ps = src;

    if (dst < src) {
        for (--pd, --ps; n; n--)
            *++pd = *++ps;
    } else if (dst > src) {
        for (pd += n, ps += n; n; n--)
            *--pd = *--ps;
    }
    return dst;
}
```

前置デクリメントと同様にrep stos命令を使った効率の良いコードになっているね(C++ Builderの-O2オプションではzrclr1関数のループが6命令，zrclr2関数が4命令で，後置デクリメントは苦手なようだ)．

A君 ：はい．

## ポインタの前置/後置インクリメント，前置/後置デクリメント

N先輩：今度はポインタに関して前置/後置インクリメント，前置/後置デクリメントを調べてみよう．リスト13はmemmove関数を無理やり前置/後置インクリメント，前置/後置デクリメントで記述した例だけど，リスト14(p.74)を見てみるとVC++ 1.5だとlodsb命令に落ちている*ps++以外はループ内は同じ効率になっているね．ディレクションフラグをセットすれば*ps--でもlodsb命令が使えるんだけど，そこまではめんどうをみてくれないようだね．

A君 ：そうなんですね．

N先輩：VC++ 5.0の場合をリスト15(p.75)(最大限の最適化：/c /Ox)に示すよ．VC++ 5.0ではlodsb，stosb命令ともに使われてないね．VC++ 4.0だと，ループ内はどれも6命令で同じ効率になっていたんだけど，VC++ 5.0は++のループ内が5命令になって，効率が良くなったんだよ(C++ Builderの-O2オプションではループ内はどれも7命令で同じ効率になった)．

### おわりに

本章ではコーディングの違いによって，最適化がどのように影響を受けるのかを見てみました．最近のコンパイラの最適化はかなり良くなってきているので，基本的にはプログラムが読みやすくなるように書けば良いと思います．しかし，回数が多いループ中などでは気になることもあります．そのときは本章で行ったようにアセンブリリストを生成して自分の目で確認してみてください．

なかしま・のぶゆき (株)Unix

〔リスト 14〕リスト 13のアセンブリリスト(VC++ 1.5, コマンドラインオプション/c /Ox)

```
;|*** void *memmove1(void *dst, const void *src, unsigned n)
;|*** {
    ..................
;   dst = 4
;   src = 6
;   n = 8
;   register di = pd
;   register si = ps
    *** 000005  8b 4e 04         mov cx,WORD PTR [bp+4]  ;dst
    *** 000008  8b 56 06         mov dx,WORD PTR [bp+6]  ;src
;|***     char *pd = dst;
; ライン 3
    *** 00000b  8b f9            mov di,cx
;|***     const char *ps = src;
; ライン 4
    *** 00000d  8b f2            mov si,dx
;|***
;|***     if (dst < src) {
; ライン 6
    *** 00000f  3b f1            cmp si,cx
    *** 000011  76 11            jbe $I180
    *** 000013  8b 56 08         mov dx,WORD PTR [bp+8]  ;n
;|***         for (; n; n--)
; ライン 7
    *** 000016  0b d2            or  dx,dx
    *** 000018  74 28            je  $I184
                  $F181:
;|***             *pd++ = *ps++;
; ライン 8
    *** 00001a  ac               lodsb
    *** 00001b  88 05            mov BYTE PTR [di],al
    *** 00001d  47               inc di
    *** 00001e  4a               dec dx
    *** 00001f  75 f9            jne $F181
    *** 000021  eb 1f            jmp SHORT $I184
    *** 000023  90               nop
;|***     } else if (dst > src) {
; ライン 9
                  $I180:
    *** 000024  3b d1            cmp dx,cx
    *** 000026  73 1a            jae $I184
    *** 000028  8b 56 08         mov dx,WORD PTR [bp+8]  ;n
;|***         for (pd += n - 1, ps += n - 1; n; n--)
; ライン 10
    *** 00002b  8b c2            mov ax,dx
    *** 00002d  48               dec ax
    *** 00002e  03 f8            add di,ax
    *** 000030  8b c2            mov ax,dx
    *** 000032  48               dec ax
    *** 000033  03 f0            add si,ax
    *** 000035  0b d2            or  dx,dx
    *** 000037  74 09            je  $I184
                  $F186:
;|***             *pd-- = *ps--;
; ライン 11
    *** 000039  8a 04            mov al,BYTE PTR [si]
    *** 00003b  4e               dec si
    *** 00003c  88 05            mov BYTE PTR [di],al
    *** 00003e  4f               dec di
    *** 00003f  4a               dec dx
    *** 000040  75 f7            jne $F186
;|***     }
;|***     return dst;
; ライン 13
                  $I184:
    *** 000042  8b c1            mov ax,cx
;|*** }
```

```
    ..................
;|*** void *memmove2(void *dst, const void *src, unsigned n)
;|*** {
    ..................
;   src = 6
;   n = 8
;   register di = pd
;   register si = ps
;   dst = 4
    *** 00004f  8b 4e 04         mov cx,WORD PTR [bp+4]  ;dst
    *** 000052  8b 56 06         mov dx,WORD PTR [bp+6]  ;src
;|***     char *pd = dst;
; ライン 18
    *** 000055  8b f9            mov di,cx
;|***     const char *ps = src;
; ライン 19
    *** 000057  8b f2            mov si,dx
;|***
;|***     if (dst < src) {
; ライン 21
    *** 000059  3b f1            cmp si,cx
    *** 00005b  76 15            jbe $I199
;|***         for (--pd, --ps; n; n--)
; ライン 22
    *** 00005d  4f               dec di
    *** 00005e  4e               dec si
    *** 00005f  8b 56 08         mov dx,WORD PTR [bp+8]  ;n
    *** 000062  0b d2            or  dx,dx
    *** 000064  74 24            je  $I203
                  $F200:
;|***             *++pd = *++ps;
; ライン 23
    *** 000066  46               inc si
    *** 000067  47               inc di
    *** 000068  8a 04            mov al,BYTE PTR [si]
    *** 00006a  88 05            mov BYTE PTR [di],al
    *** 00006c  4a               dec dx
    *** 00006d  75 f7            jne $F200
    *** 00006f  eb 19            jmp SHORT $I203
    *** 000071  90               nop
;|***     } else if (dst > src) {
; ライン 24
                  $I199:
    *** 000072  3b d1            cmp dx,cx
    *** 000074  73 14            jae $I203
    *** 000076  8b 56 08         mov dx,WORD PTR [bp+8]  ;n
;|***         for (pd += n, ps += n; n; n--)
; ライン 25
    *** 000079  03 fa            add di,dx
    *** 00007b  03 f2            add si,dx
    *** 00007d  0b d2            or  dx,dx
    *** 00007f  74 09            je  $I203
                  $F205:
;|***             *--pd = *--ps;
; ライン 26
    *** 000081  4e               dec si
    *** 000082  4f               dec di
    *** 000083  8a 04            mov al,BYTE PTR [si]
    *** 000085  88 05            mov BYTE PTR [di],al
    *** 000087  4a               dec dx
    *** 000088  75 f7            jne $F205
;|***     }
;|***     return dst;
; ライン 28
                  $I203:
    *** 00008a  8b c1            mov ax,cx
;|*** }
```





**New Products ── Intel, 低価格帯のノート PC 向け CPU「Celeron M プロセッサ」3 種類を発売**
米 Intel 社は，低価格帯のノート PC 向けに「Celeron M プロセッサ 1.30GHz」,「(同)1.20GHz」,「超低電圧版(同)800MHz」を発売した．価格はそれぞれ¥14,540，¥11,610，¥17,470である．

〔リスト 15〕リスト 13のアセンブリリスト(VC++ 5.0, コマンドラインオプション/c /Ox)

```
_dst$ = 8
_src$ = 12
_n$ = 16
_memmove1 PROC NEAR

; 3    :     char *pd = dst;
; 4    :     const char *ps = src;
; 5    :
; 6    :     if (dst < src) {

  00000 8b 54 24 08  mov     edx, DWORD PTR _src$[esp-4]
  00004 56           push    esi
  00005 57           push    edi
  00006 8b 7c 24 0c  mov     edi, DWORD PTR _dst$[esp+4]
  0000a 3b fa        cmp     edi, edx
  0000c 8b cf        mov     ecx, edi
  0000e 73 18        jae     SHORT $L119

; 7    :         for (; n; n--)

  00010 8b 74 24 14  mov     esi, DWORD PTR _n$[esp+4]
  00014 85 f6        test    esi, esi
  00016 74 2b        je  SHORT $L96
  00018 2b d7        sub     edx, edi
$L89:

; 8    :             *pd++ = *ps++;

  0001a 8a 04 0a     mov     al, BYTE PTR [edx+ecx]
  0001d 88 01        mov     BYTE PTR [ecx], al
  0001f 41           inc     ecx
  00020 4e           dec     esi
  00021 75 f7        jne     SHORT $L89

; 12   :     }
; 13   :     return dst;

  00023 8b c7        mov     eax, edi

; 14   : }

  00025 5f           pop     edi
  00026 5e           pop     esi
  00027 c3           ret     0
$L119:

; 9    :     } else if (dst > src) {

  00028 76 19        jbe     SHORT $L96

; 10   :         for (pd += n - 1, ps += n - 1; n; n--)

  0002a 8b 74 24 14  mov     esi, DWORD PTR _n$[esp+4]
  0002e 85 f6        test    esi, esi
  00030 8d 4c 3e ff  lea     ecx, DWORD PTR [esi+edi-1]
  00034 8d 54 16 ff  lea     edx, DWORD PTR [esi+edx-1]
  00038 74 09        je  SHORT $L96
$L94:

; 11   :             *pd-- = *ps--;

  0003a 8a 02        mov     al, BYTE PTR [edx]
  0003c 88 01        mov     BYTE PTR [ecx], al
  0003e 49           dec     ecx
  0003f 4a           dec     edx
  00040 4e           dec     esi
  00041 75 f7        jne     SHORT $L94
$L96:

; 12   :     }
; 13   :     return dst;

  00043 8b c7        mov     eax, edi

; 14   : }
    ..................
_dst$ = 8
_src$ = 12
_n$ = 16
_memmove2 PROC NEAR

; 17   : {

  00050 55           push    ebp

; 18   :     char *pd = dst;
; 19   :     const char *ps = src;
; 20   :
; 21   :     if (dst < src) {

  00051 8b 6c 24 08  mov     ebp, DWORD PTR _dst$[esp]
  00055 56           push    esi
  00056 57           push    edi
  00057 8b 7c 24 14  mov     edi, DWORD PTR _src$[esp+8]
  0005b 3b ef        cmp     ebp, edi
  0005d 73 20        jae     SHORT $L124

; 22   :         for (--pd, --ps; n; n--)

  0005f 8b 74 24 18  mov     esi, DWORD PTR _n$[esp+8]
  00063 8d 4d ff     lea     ecx, DWORD PTR [ebp-1]
  00066 85 f6        test    esi, esi
  00068 8d 57 ff     lea     edx, DWORD PTR [edi-1]
  0006b 74 2c        je  SHORT $L112
  0006d 2b d1        sub     edx, ecx
$L105:

; 23   :             *++pd = *++ps;

  0006f 8a 44 0a 01  mov     al, BYTE PTR [edx+ecx+1]
  00073 41           inc     ecx
  00074 4e           dec     esi
  00075 88 01        mov     BYTE PTR [ecx], al
  00077 75 f6        jne     SHORT $L105

; 27   :     }
; 28   :     return dst;

  00079 8b c5        mov     eax, ebp

; 29   : }

  0007b 5f           pop     edi
  0007c 5e           pop     esi
  0007d 5d           pop     ebp
  0007e c3           ret     0
$L124:

; 24   :     } else if (dst > src) {

  0007f 76 18        jbe     SHORT $L112

; 25   :         for (pd += n, ps += n; n; n--)

  00081 8b 74 24 18  mov     esi, DWORD PTR _n$[esp+8]
  00085 85 f6        test    esi, esi
  00087 8d 14 2e     lea     edx, DWORD PTR [esi+ebp]
  0008a 8d 0c 3e     lea     ecx, DWORD PTR [esi+edi]
  0008d 74 0a        je  SHORT $L112
$L110:

; 26   :             *--pd = *--ps;

  0008f 8a 41 ff     mov     al, BYTE PTR [ecx-1]
  00092 49           dec     ecx
  00093 4a           dec     edx
  00094 4e           dec     esi
  00095 88 02        mov     BYTE PTR [edx], al
  00097 75 f6        jne     SHORT $L110
$L112:

; 29   : }
```

## 第3章

ライブラリにしてブラックボックス化する

# 関数作成の勘所

中島 信行

C言語のプログラムは関数が基本単位となっており，関数を寄せ集めて，ひとつのプログラムを構築していきます．簡単なプログラムであればmain関数だけで作成することもありますが，役に立つ処理であれば，たとえわずか数行の短いプログラムであっても，独立した関数として記述しておけば(さらに，ライブラリとして登録しておけば)別のプログラムを作成するときにも利用可能となります．

C言語では関数が比較的手軽に作成できるため，深く考えずに関数を作成することも少なくありません．しかし，関数の作成にあたって自分なりの指針をもつようにすれば，プログラミングの上達が早くなるものと思います．それではC言語の関数に関して初心者B君とN先輩の会話をお楽しみください． (筆者)



## 関数の基本はブラックボックス化すること

### ● 関数の作成方針はどのように決めるか

B君　：先輩，おはようございます．

N先輩：おはよう．B君もCでプログラムをけっこう書いてるから，Cの関数について感じることはあるかい．

B君　：とくに何も考えずに作ってますけど．

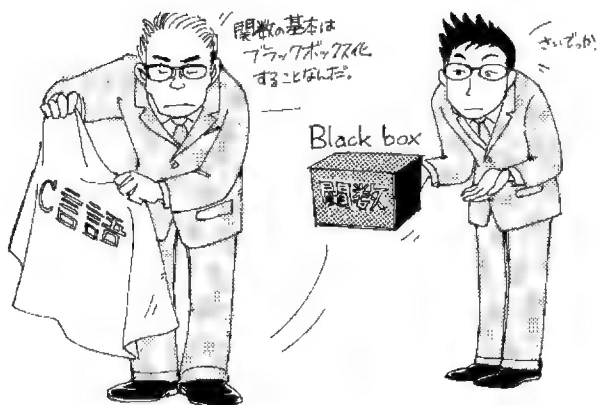
N先輩：それはまずいね．プログラムというのはそれなりの方針をもって組まないと保守しやすいものができないんだよ．

B君　：あっ，そうでしょうね．

N先輩：関数の作成方針は人それぞれに異なっていて，ある程度というか，かなりというか，本人の嗜好的な要素が加わる場合も多いから，すべての人に適用できる方針はないかもしれないけど．たとえば，

(1) 一画面(25行～50行程度)以内に収める
(2) 行数を気にせずに機能単位でまとめる
(3) 複数の箇所で呼び出すものを中心にして関数にする

などが挙げられるね．




B君　：あっ，そういうことですか．それなら，僕は機能単位でまとめていますよ．

N先輩：あっ，そうかい．(1)はエディタで編集することを考慮していて，大昔はラインプリンタ連続用紙の60行程度，エディタが便利になってきた20年くらい前は25行程度を目安にしていたんだ．最近だと，Windowsのエディタの画面の行数が目安になるんじゃないかな．一画面にはいるということが行数の基準だから，自分の使用している環境にある程度左右されると思うよ．

B君　：でも，他の人もソースを見ることがあるでしょうから，他の人の環境もある程度考えないといけないですね．

N先輩：そういうことだね．B君のやってる(2)はもっともオーソドックスな考え方だね．人間は一度にたくさんのことを考えるのはあまり得意じゃないから，一つの関数で一つの機能を記述するようにして，まとめていけば，関数の作成やデバッグがやりやすくなるんだ．

B君　：そうですね．

N先輩：(3)は関数のサブルーチンとしての性格に重点を置いたものだよ．複数箇所で使用する部分を関数にしておけばコード効率が良くなるし，修正なんかのときにも一か所ですむから効率が良いんだ．最近はエディタが親切だから，似たような処理の部分をエディタでとってきて一部だけ変えるってことを初心者はよくやるけど，あまり感心しないね．

B君　：えっ，僕もよくやりますけど．

N先輩：プログラムを作成する前には詳細仕様書を書くんだけど，この段階でどことどこが似たような処理になるか目星を付けておいて，共通の関数にまとめるようにしなくちゃいけないんだよ．

B君　：でも，一部の処理が違うから，同じ関数にはできないんじゃないですか．

N先輩：一部が違ってても，関数の引き数で一部の処理を切り分けることができるじゃないか．

B君　：あっ，そうですね．

N先輩：簡単なプログラムだと，詳細仕様書を書かずに概略仕様書に少し手を加えた程度のもので，プログラムを書くこともあるけど，このときでも，似たような箇所があれば，エディタで取ってくる前に関数化することを検討すべきなんだ．似たような処理があちこちに散らばってると，仕様追加で修正するときにたいへんなんだよ．

B君　：今後，改めます．

N先輩：複数の人でプログラムを組むときは各個人が似たような関数を作ってしまうことが多いけど，こんなときはリーダの人がうまくまとめてやらないといけないんだ．この辺の話はB君にとっては数年後のことだから，またの機会にしよう．

B君　：はい．

● 関数の基本――ブラックボックス化すること

N先輩：さっき話した以外にもいくつか考えられるだろうし，それらのうちのいくつかを組み合わせて考慮する場合もあるだろうけど，一つだけ言えることは「関数の基本はブラックボックス化すること」なんだ．パソコンのROM BIOS，MS-DOSのDOSファンクション，WindowsのAPIなどがハードウェアの詳細をブラックボックス化しているように，入出力パラメータをはっきりさせて，その関数で「何ができるのか」，「どういう処理がされているのか」というようなことをはっきりさせることが重要なんだ．

B君　：言われてみればそんな気がしますね．

N先輩：関数の仕様書を別に作成するって方法をとることもあるけど，Cのライブラリ関数のように仕様が固まった関数は別として，仕様追加が行われるプログラムだと，関数は生きてるから，共通部分を切り出してまとめておいた関数も変わる可能性があるよね．だからソースリストの始めに入出力と処理してる機能を箇条書きにしておくってのが，実際的な運用法になる場合も少なくないんだ．

B君　：そうなんですか．

N先輩：仕様書が別ファイルになっていると，一覧するときには便利なんだけど，プログラムを修正したときに仕様書まで修正が反映されるってことは期待できないことが多いんだよ．

B君　：そうかもしれないですね．ところで，先輩はどんな基準をもってるんですか．

N先輩：僕は関数の行数はそれほど気にしていないんだよ．関数を作成する際の僕なりの基準は，

〔図1〕一つのファイルの行数が多くなっても無理やり別ファイルに分けない

```
static int flag;

void func1(void)
{
    if (flag)
        ...;
    ...;
}
```

```
void func2(void)
{
    if (.....)
        flag = TRUE;
    ...;
}
```

(1) 機能単位にまとめる

(2) 行数が多く(たとえば数百行に)なる場合はサブファンクションごとに関数を分けることを検討する．ただし，各サブファンクションが密接にかかわっている場合は行数は無視して，一つの関数にする．たとえば，switch文の場合分けが多くて行数が多くなるときは各caseに対する処理がプログラムを読むための単位になるから，無理して関数に分けないようにする

(3) 複数ファイルから構成される場合には，他のファイルから参照しない関数や変数などはstaticにする

(4) 他の関数で参照しない変数は関数外で定義する外部変数にしない．関数内部で静的変数(static)や自動変数(auto)として定義する

というような感じかな．その昔のデバッガがSYMDEB(シンボリックデバッガ)の時代には変数を参照するためにわざと外部変数にすることがあったけど，最近のソースレベルのデバッガだと局所変数(関数内部の変数)は自動でローカルウィンドウ内に表示されるようになったから，かえって局所変数のほうがデバッグ時に楽だったりするよ．(3)と(4)は数百行程度のプログラムだったら大勢にそれほど影響しないけど，数万行以上のプログラムだと仕様変更時にけっこう役に立つもんだよ．プログラムは作成した人間が保守するとは限らないし，時間がたてば別人が書いたものとほとんど変わらなくなることもあるから，保守に力点をおいて作成することが大切だと思うよ．

● 1ファイルの行数

N先輩：似たようなことに一つのファイルの行数をどの程度にするかってのがあるけど，こちらも基本的には機能単位にまとめるってことで行数はあまり意識していないよ．ライブラリ的な関数だと，それだけで独立しているから，1関数1ファイルにして，ライブラリに登録するけど，アプリケーションは行数が多いからといって無理やり分けるとstaticにできる変数を外部変数にしなくちゃいけなくなったりするからね(図1)．

## Call by value とCall by reference の違い

### ●「Call by value(値による呼び出し)」のときの注意

N先輩：ところでB君は「Call by value」と「Call by reference」というのを知ってるかい．

B君　：どっかで聞いたことがありますけど，はっきりとはわかりません．

N先輩：何も難しいことじゃないんだよ．Cの関数は基本的に引き数が値で受け渡しされるんだ．これを「Call by value(値による呼び出し)」と呼ぶんだ．だから，呼び出された関数側で引き数の内容を変更しても，呼び出し元の変数には影響が及ばないんだ．たとえば，リスト1は階乗計算のプログラムだけど，この関数を，

```
fact(val);
```

のように呼び出すと，valの値が一時的なコピーとしてスタックに積まれて，関数factに渡されるんだ(図2)．関数fact内で実引き数valに相当する仮引き数ulvalを変更しても，実引き数valには影響が及ばないんだ．だから，リスト1はリスト2のように，引き数をそのままワーク変数としてより効率的に書き直すことができるんだよ．だけど，配列は例外で，引き数として渡されると配列の先頭のアドレスが渡されるから，関数内で変更すると呼び出し元の変数が書き換えられてしまうんだ．

B君　：ということは，呼び出し元の変数を変更しようと思えば変数のアドレスを渡せば良いということですよね．

N先輩：そうだね．

### ●「Call by reference(参照による呼び出し)」のときの注意

N先輩：「Call by value」に対して値ではなくて，その変数のアドレスを渡す呼び出しを「Call by reference(参照による呼び出し)」と呼ぶんだ．Cだとどちらの呼び出しも可能だけど，昔の古いFORTRANのように「Call by reference」しかできない言語もあるんだ(もっとも，FORTRANも僕がやってた時代から大きく成長してるので，今だとできるみたいなのだが)．リスト2を無理やりアドレス渡しのプログラムで記述するとリスト3のようになるのはわかるよね(図3)．

B君　：なんとなくわかります．

N先輩：これを呼び出すときには，

```
fact(&ulval, &ulfact);
```

〔リスト1〕階乗計算その1

```
unsigned long fact(unsigned long ulval)
{
    unsigned long ul;
    unsigned long ulfact;

    for (ulfact = 1L, ul = ulval; ul; --ul)
        ulfact *= ul;
    return ulfact;
}
```

〔リスト2〕階乗計算その2

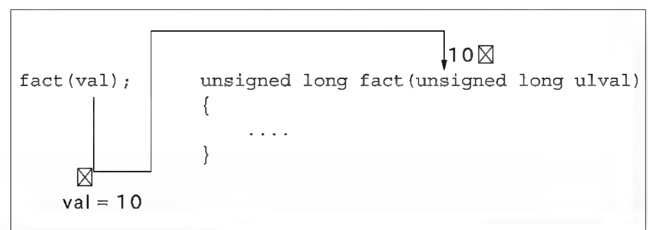
```
unsigned long fact(unsigned long ulval)
{
    unsigned long ulfact;

    for (ulfact = 1L; ulval; --ulval)
        ulfact *= ulval;
    return ulfact;
}
```

〔リスト3〕階乗計算その3

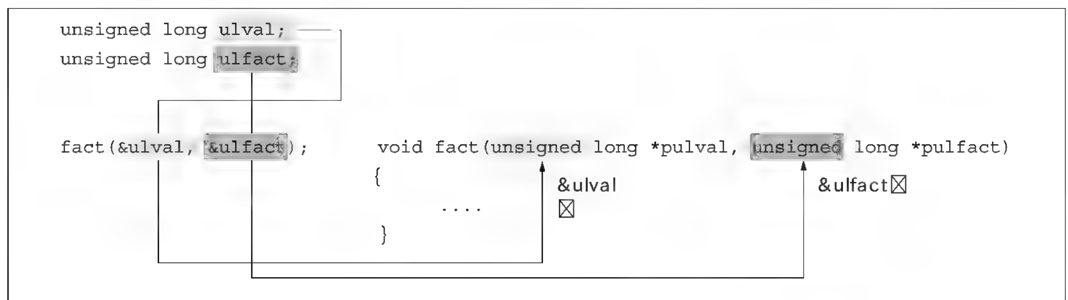
```
void fact(unsigned long *pulval, unsigned long *pulfact)
{
    for (*pulfact = 1L; *pulval; --*pulval)
        *pulfact *= *pulval;
}
```

〔図2〕Call by value



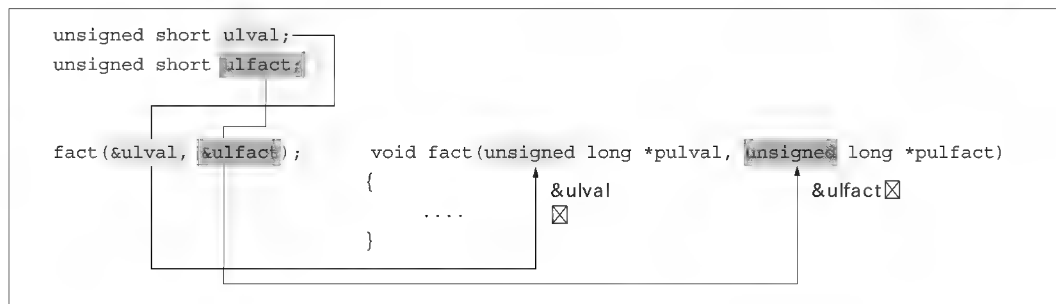
「Call by value」では変数valの内容(=10)が渡される

〔図3〕Call by reference



「Call by reference」では変数のアドレスが渡される

〔図4〕
Call by referenceで引き数の型が違う場合




関数factではunsigned longで計算するため，ulvalとulfactの直後に割り付けられている2バイトが破壊される

というように引き数として変数のアドレスを渡す形式になるんだ．ulvalが階乗を求めたい入力変数で，ulfactが階乗計算の結果を格納する変数だけど，リスト3だと計算を終えて呼び出し元に戻ると，入力変数のulvalはかならず0に破壊されてしまうんだ．

B君　：そうですね．

N先輩：ここで質問だ．もし，呼び出し時のulvalやulfactが間違えてunsigned short（2バイト）だった場合を考えてみてよ（図4）．

B君　：リスト3だとunsigned long（4バイト）として計算するから，これらの変数の直後の2バイトを破壊してしまうことになりますよね．

N先輩：そうだね．自分とまったく関係のない変数が関数呼び出しによって破壊されてしまうわけだよ．もし，ulvalとulfactの領域が連続していれば，計算のたびに破壊されて正しい結果も得られないよね．場合によってはループの抜け出し条件が永久に満足できなくて，無限ループになってしまう可能性もあるんだ．

B君　：でも，そんな間違いは普通しないんじゃないですか．

N先輩：まあ，そうだけど，最初の基本設計で手を抜くと，途中で関数の仕様を変更する必要が生じてきて，修正漏れで見のがしてしまうということも考えられるんだ．ANSI Cに準拠したCコンパイラだとこのような間違いを記述すると，正しいプロトタイプ宣言があれば警告やエラーが表示されるけど，警告を無視して（見過ごして）オブジェクトを作成することもできるよね（ANSI C以前のCコンパイラでは警告も表示されずにオブジェクトが作成される）．

B君　：急いでるときなんかは，警告を無視するときもありますよね．

N先輩：リスト2だと，正しいプロトタイプ宣言があればunsigned shortの値がunsigned longに拡張されて渡されるだけだから，まったく問題は生じないよね．この例のように「Call by reference」にはつねに危険がともなうんだ．リスト2とリスト3ではどちらがブラックボックスと呼べるかを考えれば，関数はできるだけ「Call by value」で記述するほうが安全で独立性の高い（ブラックボックスと呼べる）関数になることがわかると思うけど．

B君　：はい．

## C言語のライブラリに学ぶ

### ● ライブラリ関数の作成にあたっての注意

N先輩：Cは構文が非常に簡単だから，実質的にはライブラリ関数が使い勝手を左右するよね．どんなにすばらしい最適化を行うCコンパイラでもライブラリ関数がなければ非常に使いにくいだろうね．25年以上も前のことだけど68K系のクロスコンパイラを購入したときにmemcpyやstrcpyなどのもっとも基本的な標準ライブラリも付属していなくて唖然としたことがあるんだ．当然，使う前に必要な関数を自作したのはいうまでもないけどね．最近ではそういうことはないけど．

B君　：ライブラリがなかったらたいへんですよね．

N先輩：そういう意味で改めてライブラリ関数を眺めてみると，やはり非常によくできてるよ．だから，関数の作成にあたってはライブラリ関数を参考にするのが良いと思うよ．関数の作成にあたって，ポイントとなる点には

（1）関数の仕様
（2）関数名
（3）引き数
（4）引き数の順番
（5）返り値

などが挙げられるよね．これらを決めるのにライブラリ関数が参考になる場合も少なくないんだ．その昔，似たようなライブラリを探していると，すでにライブラリ関数にあったなんてこともあったよ．

B君　：ライブラリ関数が全部頭に入ってる人はいないでしょうから，そういうこともあるでしょうね．

N先輩：それじゃ，簡単な例を示してみようか．整数を文字列に変換する用途はけっこうあるよね．こういうときには通常sprintf関数などを使って変換するよね．printf系の関数は多機能で非常に便利なんだけど，FA分野で使用するときには注意が必要なんだ．

B君　：どんな注意ですか．

N先輩：MS-C，VC++ なんかだとスタックチェックのコードが含まれていておもしろくないんだ．組み込み機器でリアルタイムモニタを使うとスタックをタスクごとに別々に用意するから，MS-Cが仮定しているスタック領域と違ってきて，スタックオーバフローのエラーになるんだよ．

B君　：それはおもしろくないですね．

**● ライブラリ関数から新しい関数を作る**

N先輩：そこで，ライブラリ関数を当たってみると，

```
char *ltoa(long value, char *str,
                                 int base);
```

value　：変換値
str　　：変換後の文字列格納バッファ
base　：基数（2～36）

という関数があるんだ．この関数はlongを文字列に変換してくれて，baseが10のときは符号付きの変換を行って，10以外の基数のときには符号なしの変換をやってくれるんだ．でも，この関数は%ld，%lxの変換に相当する機能を提供するだけで変換する文字列の桁数を指定できないんだ．そこで，ltoa関数を参考にして変換する文字列の桁数を指定できる関数を考えてみよう．B君だったら，まず関数名はどうする？

B君　：longの値を $n$ 桁のASCII文字列に変換するということでltonaってのはどうですか．

N先輩：けっこう良いネーミングだと思うよ．次に，引き数だけど．

B君　：引き数は桁数の指定を追加するだけで良いですよね．

N先輩：そうだね．その追加した桁数は引き数のどの位置にすればいいかな．

B君　：やっぱり追加したんだから，最後に追加するのが良いと思いますけど．

N先輩：そういう考え方も成り立つだろうね．順番に考えていってみようか．引き数の追加位置は，

先頭
valueの次
strの次
最後

の4か所の候補があるよね．

B君　：はい．

N先輩：まずは先頭位置だけど，long→文字列変換（valueを変換してstrに格納）という流れを考えると，先頭位置というのは問題外だよね．

B君　：僕もそう思います．

N先輩：この関数の主役である入力と出力を考えるとvalueとstrだから，valueの次（valueとstrの間）の位置に追加するというのも違和感があるよね．

B君　：はい．

N先輩：消去法で二つの候補が消えたから，残りは二つだけど，今度はstrにより密接な引き数は桁数と基数のどちらか考えてごらん．

B君　：strはポインタですから，そのサイズである桁数のほうがより密接な関係にあると思います．

N先輩：僕もそう思うね．ということで，密接な関係にある引き数を並べるという方針でstrの次の位置というのがより自然だと思うよ．B君が最初に言ったように，追加するんだから引き数の最後で指定するという案も考えられるけどね．変換基数を指定するライブラリ関数を並べてみると，

```
long strtol(const char *str,
                 char **endptr, int base);
unsigned long strtoul(const char
          *str, char **endptr, int base);
char *itoa(int value,
                   char *str, int base);
char *ltoa(long value,
                   char *str, int base);
char *ultoa(unsigned long
             value, char *str, int base);
```

となるけど，何か気づいた点はないかい．

B君　：どれも基数baseが最後にきていますね．

N先輩：ということで，追加した桁数は最後の引き数である基数の手前という覚え方ができるだろ．

B君　：そう言われてみるとstrの次の位置というのがベストだという気がしてきましたよ．

N先輩：B君は暗示にかかりやすいんだね．ともかく，桁数はstrの次の位置に追加することにしよう．返り値は使われることは少ないからvoidでも良いんだけど，ltoa関数にあわせて変換後の文字列を格納するstrを返すことにしよう．

B君　：仕様を変えると間違えやすいですからそれが良いのでしょうね．

N先輩：最後に肝心の関数の仕様だけど．

B君　：これもltoa関数にあわせて，基数として2～36を指定して，基数が10で変換値valueが負のとき符号付きとみなすようにすれば良いんじゃないですか．

N先輩：ltoa関数と仕様を変えると間違いやすいから同じ仕様にするというのがもっとも妥当な仕様だろうね．この関数だと基数baseの仕様がポイントになると思うんだ．たとえば，負の基数を認めて，負の基数のときには変換値valueを符号付きとみなして，正の基数のときには符号なしとみなすという仕様も考えられると思うんだ．この仕様にすると一つの関数で符号付き整数と符号なし整数を処理することができるようにな

るから ultona という unsigned long を処理する関数を作らなくても良くなるから便利なんだ(人によっては符号なし整数を処理する ultona に分けたほうが良いという人もいると思うけど). ltoa 関数のように10進だけ特別扱いしていないという自然さもあるしね. ということで, 今回は基数の正負で符号付きと符号なしを指定する仕様にしたのがリスト4(ltona.c)だよ. 関数は引き数の仕様を間違えたときでも, できるだけ無限ループや暴走をしないようにしておくというのが基本だから, 基数が±2～36以外のときの対処もポイントの一つになるだろうね. 基数の範囲をチェックしなかったら基数に間違えて1を指定すると do～while のブロックが無限ループになるだろ.

B君 : そうですね.

N先輩 : リスト4だと範囲外の基数が指定されるとすべてスペースがセットされることになってるけど全部'*'印にして目立たせるというのもおもしろいかもしれないね. この部分は間違いを知らせるだけだから, 無限ループにならなければどちらでも良いと思うけど. 基数の間違いを知らせるために基数が±2～36以外のときはエラーとしてNULLを返すという案も考えられるけど, 基数の範囲は実行時にチェックする部分じゃないし, 最終的には直さなくてはいけないものだから, デバッグ中に表示を見てすぐわかるほうが良いだろうね.

B君 : 単純な関数ですけど, 細かく見ていくといろいろとあるんですね.

## 汎用性のあるものは別ファイルにしてライブラリに登録

N先輩 : B君は自作のライブラリ関数をもってるかい.

B君 : いいえ. 僕は必要な部分だけをエディタで取ってきてプログラムの中に入れて使ってますけど.

N先輩 : 自作の関数の中で汎用性が高いものは別ファイルにしておいて, 自作ライブラリに登録するのが便利だよ.

B君 : ライブラリを一つのソースにまとめておくというのはどうですか.

N先輩 : 一つのソースだとリンクしたときに不要なものもくっついちゃうから, あまりおもしろくないね. 別ファイルにしてライブラリに登録しておくとプロトタイプ宣言を変更できるというメリットもあるんだ. もっとも代表的なのは printf のように引き数が可変の関数なんだけど,

```
int func(char *arg1, ...);
```

というプロトタイプ宣言と,

```
int func(char *arg1, int arg2,
                        long arg3)
{
    /* ... */
}
```

という関数本体が同一ファイルにあるとコンパイルエラーになる処理系も, その昔, あったしね.

B君 : そうなんですか.

N先輩 : ちょっと変わった例としてリスト5とリスト6を見くらべてごらん.

B君 : これは何をする関数ですか.

N先輩 : これらはMS-DOSの時代の far ポインタのセグメントの正規化(オフセットアドレスが0～0x0fになるようにセグメントを調整すること)を行う関数なんだ. 別ファイルにするとリスト5のように記述して, 参照するプログラム側でプロトタイプ宣言を,

```
void far *psegnorm(void far *);
```

のようにすることができるんだ. でも, 同一ファイルに記述するとリスト6のように記述しなくてはいけな

〔リスト4〕longの値を $n$ 桁のASCII文字列に変換する関数 ltona.c

```
/*
    long→base進n桁文字列変換

    Bin. → n桁文字列 + '¥0'
            str[] サイズ >= n+1

    123  → "  123"
    0    → "    0"

    引き数 long lval;  変換数値
           char *str;  base進文字列格納アドレス (MSB→LSB : サイズ >= n+1)
           int     n;  変換桁数
           int  base;  変換基数 (±2～36, 負 : 符号付き, 正 : 符号なし)

    返り値 str
*/
char *ltona(long lval, char *str, int n, int base)
{
    static const char xdigit[] =
                            {"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"};
    unsigned long ulval;            /* 符号なし整数               */
    unsigned int  ubase;            /* 符号なし変換基数           */
    int sign = 0;                   /* 符号                       */
    char *ps = str + n;             /* ストリング最後のアドレス +1 セット*/

    *ps = '\0';                     /* ストリング ターミネータ セット   */
    if (base < 0) {                 /* 符号付き変換 ?             */
        ubase = -base;
        if (lval < 0) {
            lval = -lval;
            sign = -1;
            --n;
        }
    } else {                        /* 符号なし変換               */
        ubase = base;
    }
    ulval = lval;
    if (n > 0 && (2 <= ubase && ubase < sizeof(xdigit))) {
        do {
            *--ps = xdigit[(unsigned)(ulval % ubase)];
        } while (--n > 0 && (ulval /= ubase) != 0);
    }
    if (sign < 0)
        *--ps = '-';                /* 符号 セット                */
    while (--n >= 0)
        *--ps = ' ';                /* スペース セット            */
    return str;
}
```

くなるんだ．リスト 6だと共用体に代入するぶんだけ効率が悪くなってるよ．

B君　：そうですね．

N先輩：とくに，voidへのポインタを引き数にもつ関数ではこのようなメリットが得られことがあるよ．もっとも，リスト 7のような（リスト 5で生成されるコードを直接記述するような）トリッキーな記述法を使えば同一ファイルにしても効率は悪くならないんだけど．

## 引き数の渡し方

N先輩：B君は関数への引き数の渡し方はどうしてる？

B君　：どうしてるって，普通にしてますよ．

N先輩：引き数の渡し方には主として

（1）関数の引き数として渡す

（2）外部変数を介して渡す

（3）（1）+（2）

が考えられるよね（細かくいえば（1）はさらに,「Call by value」と「Call by reference」に分けることができるけど）．

B君　：そうですね．

〔リスト 5〕セグメントの正規化その 1

```
union upnt {
    unsigned long  ladr;
    unsigned short uadr[2];
    void far *padr;
};

void far *psegnorm(union upnt up)    /* セグメント正規化 */
{
    up.uadr[1] += up.uadr[0] >> 4;
    up.uadr[0] &= 0x0f;
    return up.padr;
}
```

〔リスト 6〕セグメントの正規化その 2

```
void far *psegnorm(void far *p) /* セグメント正規化 */
{
    union upnt {
        unsigned long  ladr;
        unsigned short uadr[2];
        void far *padr;
    } up;

    up.padr = p;
    up.uadr[1] += up.uadr[0] >> 4;
    up.uadr[0] &= 0x0f;
    return up.padr;
}
```

〔リスト 7〕セグメントの正規化その 3

```
void far *psegnorm(void far *p) /* セグメント正規化 */
{
    *((unsigned far *)&p + 1) += *((unsigned far *)&p) >> 4;
    *((unsigned far *)&p) &= 0x0f;
    return p;
}
```

N先輩：一般的には関数の独立性を高めるために（1）の形式を取るよね．

B君　：はい．

N先輩：実際，ライブラリ関数のほとんどは（1）の形式なんだ．でも，エラーを関数の返り値としてだけではなくて，エラーの詳細を外部変数のerrnoに返す関数なんかも少なくないだろ．

B君　：あっ，そうですね．

N先輩：このようにかならず参照する必要はない（必要なときにだけ参照する）値を外部変数に返すというのは，引き数を減らして関数呼び出しをシンプルにするためには有効なんだ．

### ● 複数の関数から参照する変数

N先輩：FA関係だと，いろいろな関数から参照しているフラグ変数なんかは引き数で渡さずに外部変数を介して渡すことが多いんだ（図 5）．FAなどの自動制御のプログラムだと入出力の状態を外部変数のフラグにセットしておいて，複数の関数から直接参照する場合のほうが多いんだ．このような場合，いちいち関数の引き数として渡すと引き数の数が多くなって，かえってプログラムが読みにくくなるものなんだよ．それに，関数の引き数は一般的にはスタック経由で渡されるから，呼び出しの際のオーバヘッドが大きくなって，とくにループしている場合なんかだと実行速度に影響することも考えられるんだ．

B君　：でも，外部変数で渡すのは良くないって言いますよね．

N先輩：引き数を外部変数を介して渡すことの最大の欠点は，意図しない副作用が起きやすいことなんだ．たとえば，途中で仕様を追加した場合などに，関数の入力である引き数と出力である返り値はきっちりと抑えても，外

〔図 5〕多くの関数から参照される変数は外部変数として渡す

```
static int flag;

void func1(void)
{
    if (flag)
        ...;
    ...;
}

void func2(void)
{
    if (.....)
        flag = TRUE;
    ...;
}

/*  ... */

void funcn(void)
{
    if (flag)
        ...;
    ...;
}
```

部変数を介した入出力を見落とす場合があるんだ．それに，外部変数はどの関数からも参照可能だから，仕様追加で外部変数を操作したために既存の部分が影響を受けるといった場合も考えられるんだ．

B君　：使ってる外部変数の一覧でも書いておけば良いんでしょうけど．

N先輩：そうだね．でも，その一覧がちゃんと修正ぶんもフォローされてるかどうかって点も問題だよね．この点をきっちりと抑えておくというのは口で言うのは簡単だけど意外と難しいものだよ．もっとも，最近は便利なツールがたくさんあるから，ツールを使ってクロスリファレンスを取ったりすれば，後からの仕様追加がやりやすくなることもあるけどね．

### ● 引き数が多くなる場合の対処法

N先輩：極力，関数の引き数として渡そうとすると，引き数が多くなりすぎる場合があるよね．5，6個だとたいしたことはないけど．B君は10個以上の引き数を渡す必要があるとしたらどうやって渡してる？

B君　：単純に引き数として書き並べますけど．

N先輩：10個以上になると引き数の指定もけっこうめんどうだよね．それに，引き数の指定間違いも起きやすくなるものだよ．型が違えばプロトタイプ宣言をしておけば警告が表示されるから，間違いがわかるけど，10個以上になると型が全部異なる場合のほうが少ないだろうから間違えても気づかないこともあるだろうね．

B君　：そうですね．

N先輩：こういうときには一部を外部変数で渡すというのも一つの案だけど，変数をいくつかのグループに分けて構造体として定義し直して，その構造へのポインタを引き数として渡すと関数呼び出しがスッキリする場合があるんだ（図6）．

B君　：言われてみればそうですね．

〔図6〕引き数が多い場合は構造体として渡す

```
void func(int arg1, long arg2, char arg3[], ..., int argn)
{
    /* ... */
}
```

⬇

```
struct arg {
    int  arg1;
    long arg2;
    char arg3[10];
    /* ... */
    int argn;
} args;

void func(struct arg *pargs)
{
    /* ... */
}
```

## 分岐と関数へのポインタ

N先輩：Cで分岐を行うためには，

（1）if文（if～else if～else～）
（2）switch文
（3）関数へのポインタの配列
などが使用できるよね．

B君　：はい．

N先輩：簡単な分岐だと，if文で済ませられるけど，分岐の場合分けが多くなると，switch文を使うんだ．場合分けがどの程度になれば，switch文が適しているかという点は多少趣味的な要素もあるから，意見の分かれるところだけど，3～5分岐がif文とswitch文の境目といったところかな（異論のある方もいるかもしれないが）．

B君　：そんなもんですかね．

N先輩：後から追加する可能性があるときには，分岐が二つでも，switch文にしておくってこともあるけどね．各分岐の処理の呼び出し頻度に極端な差がある場合なんかには，あえてif文を使用して頻度の高い順にif文で条件分岐させる場合もあるんだよ（リスト8）．この条件分岐がループしている場合には実行速度に若干の差が生じることもあるんだ．

〔リスト8〕頻度の違う条件

```
{
    if (case0) {
        ;   /* もっとも頻度の高い処理   */
    } else if (case1) {
        ;
    } else if (case2) {
        ;

    /*  ... */

    } else {
        ;   /* もっとも頻度の低い処理   */
    }
}
```

〔リスト9〕switch文

```
void func(int n)
{
    switch (n) {
    case  0 :
        /* 処理 0   */
        break;
    case  1 :
        /* 処理 1   */
        break;
    case  2 :
        /* 処理 2   */
        break;
    case  3 :
        /* 処理 3   */
        break;
    case  4 :
        /* 処理 4   */
        break;
    case  5 :
        /* 処理 5   */
        break;

        /* ...  */

    default :
        break;
    }
}
```

〔リスト 10〕関数へのポインタの配列（分岐条件が連続している場合）

```
void func0(void)    /* 処理 0   */
{
    /* ... */
}

void func1(void)    /* 処理 1   */
{
    /* ... */
}

void func2(void)    /* 処理 2   */
{
    /* ... */
}

void func3(void)    /* 処理 3   */
{
    /* ... */
}

void func4(void)    /* 処理 4   */
{
    /* ... */
}

void func5(void)    /* 処理 5   */
{
    /* ... */
}

/*  ... */

void funcdef(void)  /* 処理 : デフォルト */
{
    /* ... */
}

void func(int n)
{
    static void (*functbl[])(void) = {
        func0,      /* 処理 0   */
        func1,      /* 処理 1   */
        func2,      /* 処理 2   */
        func3,      /* 処理 3   */
        func4,      /* 処理 4   */
        func5,      /* 処理 5   */
        /*  ... */
    };

    if (n < sizeof(functbl) / sizeof(functbl[0]))
        functbl[n]();
    else
        funcdef();  /* 処理 : デフォルト */
}
```

〔リスト 11〕関数へのポインタの配列（分岐条件が連続していない場合）

```
void func0(void)        /* 処理 0   */
{
    /* ... */
}

void func11(void)       /* 処理 11  */
{
    /* ... */
}

void func20(void)       /* 処理 20  */
{
    /* ... */
}

void func33(void)       /* 処理 33  */
{
    /* ... */
}

void func40(void)       /* 処理 40  */
{
    /* ... */
}

void func55(void)       /* 処理 55  */
{
    /* ... */
}

/*  ... */

void funcdef(void)      /* 処理 : デフォルト */
{
    /* ... */
}

void func(int n)
{
    static struct sfunctbl {
        int no;
        void (*functbl)(void);
    } sfunc[] = {
        {
            0,
            func0,      /* 処理 0   */
        },
        {
            11,
            func11,     /* 処理 11  */
        },
        {
            20,
            func20,     /* 処理 20  */
        },
        {
            33,
            func33,     /* 処理 33  */
        },
        {
            40,
            func40,     /* 処理 40  */
        },
        {
            55,
            func55,     /* 処理 55  */
        },

        /*  ... */

        {
            0,
            NULL,       /* テーブル終了 */
        }
    };
    struct sfunctbl *pfunc;

    for (pfunc = sfunc; pfunc->functbl != NULL ; pfunc++) {
        if (n == pfunc->no) {   /* 該当処理検索 */
            pfunc->functbl();
            return;
        }
    }
    funcdef();          /* 処理 : デフォルト    */
}
```

〔リスト 12〕関数内で機種を判断する

```
#define IS98    0   /* PC-9801シリーズ   */
#define ISDOSV  1   /* DOS/Vマシン       */
#define ISAX    2   /* AXマシン          */
#define IS31    3   /* J-3100シリーズ    */

int ismachine(void);

void fucn(int n)
{
    switch (ismachine()) {
    case IS98 :
        /* PC-9801 シリーズ の処理  */
        break;
    case ISAX :
        /* AX マシン の処理         */
        break;
    case IS31 :
        /* J-3100 シリーズ の処理   */
        break;
    default :
    case ISDOSV :
        /* DOS/Vマシンの処理        */
        break;
    }
}
```

● 関数へのポインタの配列

N先輩：switch文だと場合分けが多くなると分岐先のテーブルを作成してジャンプするコードが生成される場合があるんだ．これを直接Cでプログラムするとどうなるかわかるかい．アセンブラでプログラムを組んだことがあれば当たり前の手順なんだけど．

B君　：僕はアセンブラは使ったことがないですから．

N先輩：分岐が多い場合で，しかも分岐条件が連続している場合だと関数へのポインタの配列（分岐先のテーブル）を使用するとスッキリするものなんだ．たとえば，リスト9（p.83）はリスト10のように記述することができるよ．デバイスドライバをC言語で記述するとコマンドコードにしたがって，このような分岐になるんだ．

B君　：そうなんですか．

N先輩：if文だと後になるにしたがって，呼び出し時のオーバヘッドが大きくなるけど，関数へのポインタの配列を使用すると各処理が同じオーバヘッドで呼び出されるという特徴があるんだ．

B君　：分岐条件が連続していないときはどうなるんですか．

N先輩：分岐条件が連続していなくても，リスト11のようにすれば分岐先をテーブルにすることができるけど，ここまですると switch文とどちらがわかりやすいかは意見の分かれるところだろうね．

● 機種ごとに関数を変える

N先輩：今のWindowsパソコンは，IBM PC/AT互換機だけになってしまったけど，MS-DOSの時代には，PC-9801シリーズ，IBM PC/AT互換機，J-3100シリーズ，AXマシン，FMRなんて，いろんなパソコンがあったんだ．

B君　：少しは知ってます．

〔リスト 13〕機種ごとに関数を変える

```
#define IS98    0   /* PC-9801シリーズ   */
#define ISDOSV  1   /* DOS/Vマシン       */
#define ISAX    2   /* AX マシン         */
#define IS31    3   /* J-3100 シリーズ   */

int ismachine(void);
void func98(int);
void funcax(int);
void func31(int);
void funcpc(int);

void (*pfunc)(int);

int main(int argc, char *argv[])
{
    switch (ismachine()) {
    case IS98 :     /* PC-9801シリーズ   */
        pfunc = func98;
        break;
    case ISAX :     /* AXマシン          */
        pfunc = funcax;
        break;
    case IS31 :     /* J-3100シリーズ    */
        pfunc = func31;
        break;
    default :
    case ISDOSV :   /* DOS/Vマシン       */
        pfunc = funcpc;
        break;
    }

    /* ... */

    return 0;
}

void func98(int n)
{
    /* ... */
}

void funcax(int n)
{
    /* ... */
}

void func31(int n)
{
    /* ... */
}

void funcpc(int n)
{
    /* ... */
}
```

N先輩：組み込み機器でも，うちみたいなところは客先ごとに少しずつソフトやハードが違うっていうこともあるんだ．場合によってはハードが製造中止になって別のものになるってこともあるよね．

B君　：そうですね．

N先輩：このようなときには，さっきのパソコンの例だと，リスト12のように各関数内で機種を判断する方法もあるけど，関数へのポインタを使用するとスッキリするんだ．リスト13だと最初に機種判別を行って，関数へのポインタに各マシン用の関数をセットしているんだ．このようにすると，各機種専用のプログラムと比較してオブジェクトサイズは少し大きくなるけど，同じEXEファイルで複数のマシンで動作するように

〔リスト 14〕コマンドラインの引き数のソート

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int compar(const char **, const char **);

int main(int argc, char *argv[])
{
    if (argc >= 2) {
        qsort(argv, argc, sizeof(argv[0]),
               (int (*)(const void *, const void *))compar);
    }
    while (--argc >= 0)
        printf("%s¥n", *argv++);
    return 0;
}

int compar(const char *argv1[], const char *argv2[])
{
    return strcmp(*argv1, *argv2);
}
```

〔リスト 15〕関数へのポインタを引き数に

```
#define IS98    0   /* PC-9801シリーズ */
#define ISDOSV  1   /* DOS/Vマシン     */
#define ISAX    2   /* AXマシン        */
#define IS31    3   /* J-3100シリーズ  */

int ismachine(void);
int mainprog(int, char **, void (*)(int));
void func98(int);
void funcax(int);
void func31(int);
void funcpc(int);

int main(int argc, char *argv[])
{
    switch (ismachine()) {
    case IS98 :     /* PC-9801シリーズ  */
        return mainprog(argc, argv, func98);
    case ISAX :     /* AXマシン     */
        return mainprog(argc, argv, funcax);
    case IS31 :     /* J-3100シリーズ   */
        return mainprog(argc, argv, func31);
    default :
    case ISDOSV :   /* DOS/Vマシン      */
        return mainprog(argc, argv, funcpc);
    }
    return 0;
}

int mainprog(int argc, char *argv[], void (*func)(int))
{
    /* 実際の main プログラム */
    return 0;
}

void func98(int n)
{
    /* ... */
}

void funcax(int n)
{
    /* ... */
}

void func31(int n)
{
    /* ... */
}

void funcpc(int n)
{
    /* ... */
}
```

なって，実行ファイルの管理が楽になるんだ．それに，リスト 13だと関数呼び出し時に機種判別を行う必要がないから，通常はリスト 12よりも少し高速になるんだ．機種の自動判別が難しければ

（1）コマンドラインのオプションスイッチで指定する
（2）環境変数で指定する

などの方法も考えられるよね．

● 関数へのポインタを引き数にする

N先輩：いろいろな状態に対応するために関数へのポインタを引き数に取る場合があるよね．典型的な例がライブラリ関数の qsort なんだけど．

```
void qsort(void *base, size_t nelem,
                       size_t width,
        int (*compar)(const void *,
                      const void *));
```

base　：ソートする配列の先頭アドレス
nelem　：ソートするデータの個数
width　：データの各要素の大きさ（バイト数）
compar：比較関数へのポインタ

qsort はデータをソートして並び変える関数だけど，ソートする条件を関数へのポインタとして引き数 compar にしているんだ．リスト 14だとこの比較関数 compar を strcmp にしているけど，この関数を書き換えることで，

（1）大文字小文字を区別しないソート
（2）逆順ソート
（3）引き数がファイル名であれば拡張子によるソート

などいろいろな条件に対応できるんだ．これを応用すればリスト 13はリスト 15のように書き換えることもできるよ．

## 関数の返り値

● 関数の返り値はライブラリ関数にあわせる

N先輩：関数の返り値はときどき迷うことがあるよね．もっとも，それが求める値であれば迷うことはないけど．たとえば，リスト 4だと変換後の文字列を格納するバッファのアドレスを返しているけど，この返り値は引き数の値をそのまま返しているだけだから，使うことはあまりないよね．だから，返り値をもたない void 型の関数として記述することも考えられるよね．でも，ライブラリ関数にあわせるというのは間違いを防ぐという意味でけっこう大切なことなんだ．

B君　：そうでしょうね．

N先輩：返り値として特別な意味をもつエラーの値を振り返ってみると，

NULL　　　：ポインタを返す関数のエラー

〔リスト16〕二つ以上の返り値を返す

```
int func(int *ret1, long *ret2, ..., double *retn)
{
    int error = 0;

    /* ... */
    return error;    /* エラーの有無を返す    */
}

void mainprog(void)
{
    int ret1;
    long ret2;
    /* ... */
    double retn;

    if (func(&ret1, &ret2, ..., &retn)) {
        /* エラー */;
    }
```

－1(またはEOF)：intを返す関数のエラー
というのが一般的だよね．

B君　：そうですね．

N先輩：単純にいうと0(NULL)または－1(EOFもたいてい－1に定義されている)がエラーを意味するのが一般的なわけだ．この他にも0：正常終了，0以外：エラーといった関数も少なくない．NULLはヌルポインタでCだと無効なポインタとして使われて，他のどんなポインタの値とも区別可能なことが保証されているから，ライブラリ関数を使用している限りは0番地が有効なポインタとしてプログラムに返されることはないんだ．だから，組み込み機器の場合でもポインタを返す自作の関数のエラー値としてNULLを返せば問題が生じなくなるわけなんだ．

B君　：そうだったんですか．

N先輩：－1という値もintを返す関数でとりえない値の場合が少なくないんだ．たとえば，Cコンパイラのopen関数が返すファイルハンドル値は正常なとき正数を返すけど，ファイルハンドルが0x7fff(int：16ビット)，0x7fffffff(int：32ビット)を超えることはないとみなせるため，－1をエラー値としているわけなんだ．自作の関数でもエラー値をこのどちらかにしておけば間違いが少なくなると思うよ．

B君　：はい，そうするようにします．

N先輩：でも，正常なときでも0または－1を取る可能性があるときはなんらかの対策が必要となる場合があるよ．たとえば，ライブラリ関数でおなじみの，

```
int atoi(const char *str);
```

を見てみようか．この関数は－32768～32767のすべての値を取る可能性があるよね．だから，strの示すバッファが，

"xyz"

のように10進以外の文字列であった場合でもエラーを返すことができなくて，0を返してしまうんだ．それに，バッファが，

"654321"

のように桁溢れが生じる場合にもエラーを返すことができないんだ．通常はこの仕様を承知して使用することになるんだけど，もし，この仕様で問題があるときは，

```
long strtol(const char *str,
            char **endptr, int base);
```

str　　：変換する文字列
endptr：変換の終了位置
base　 ：基数(0，2～36)

を使用して変換を中止した文字を示すendptrを調べたり，errnoでオーバフローを調べたりするなどの対策を取らなければいけないんだ．

〔リスト17〕二つ以上の返り値を返す(構造体を使う場合)

```
struct sret {
    int *ret1;
    long *ret2;
    /* ... */
    double *retn;
};

int func1(struct sret *ret) /* リスト17     */         ┐
{                                                      │
    int error = 0;                                     ├Ⓐ ☒
                                                       │
    /* ... */                                          │
    return error;   /* エラーの有無を返す      */       ┘
}

struct sret *func2(void)    /* リスト17     */         ┐
{                                                      │
    static struct sret ret;                            ├Ⓑ
                                                       │
    /* ... */                                          │
    return &ret;                                       ┘
}

struct sret func3(void) /* リスト17  */                ┐
{                                                      │
    static struct sret ret;                            ├Ⓒ
                                                       │
    /* ... */                                          │
    return ret;                                        ┘
}

void mainprog(void)
{
    struct sret ret;
    struct sret *pret;

    if (func1(&ret)) {      /* リスト17Ⓐ  */
        /* エラー */;
    }
    pret = func2();     /* リスト17Ⓑ   */
    ret  = func3();     /* リスト17Ⓒ   */
}
```

## ● 関数の返り値は一つ

N先輩：当たり前のことだけどCの関数の返り値は一つだということも一つのポイントかもしれないね．二つ以上の値を返したい場合は，

(1) 引き数として変数のアドレスを渡す(リスト16)
(2) 返り値を構造体にまとめて引き数として構造体のアドレスを渡す(リスト17のⒶ)

(3) 返り値を構造体にまとめて構造体へのポインタを返す(リスト17のⒷ)

(4) 返り値を構造体にまとめて構造体を返す(リスト17のⒸ)

などの方法を取る必要があるんだ.(1)や(2)だと int 型の関数にして,エラーの有無を返すようにするとプログラムが見やすくなるかもしれないね.

● main関数の返り値

N先輩：ところで,B君は main 関数の返り値はなんだかわかってるかい.

B君　：えっ,とくに意識してなかったですけど.

N先輩：main 関数の返り値は int なんだ.だから,リスト18のような記述になるんだ.main 関数を呼び出すスタートアップルーチンは main 関数からの返り値(16ビットの86系では main 関数から戻ってきたときのAXレジスタの値で,68K系ではD0レジスタの値)をOSへ渡すんだ.その昔のCP/Mのように,この返り値を無視するOSもあるけど,最近のOSだとこの返り値をOSのAPIで参照することができるんだ.

B君　：そうだったんですか.

N先輩：この返り値は一般的には0が正常終了を意味し,数値が大きくなるにしたがってエラーの度合が大きくなることを意味してるものなんだ.main 関数をあえて void 型で記述する場合はリスト19のような記述をして exit 関数でOSに値を返す必要があるんだよ.ちなみにリスト20のように void 型にして明示的にOSに値を返さない場合は,main 関数の最後の時点での返り値レジスタの値(すなわちゴミ)がOSへの返り値となってしまうんだ.MAKE ユーティリティなんかだと,この返り値でエラーの有無を判断してるから,OSに値を返さないプログラムだとエラーの有無を判断できなくなってしまうんだ.

〔リスト18〕メイン関数その1

```
int main(int argc, char *argv[])
{
    /* ...... */
    return 0;
}
```

〔リスト19〕メイン関数その2

```
void main(int argc, char *argv[])
{
    /* ...... */
    exit(0);
}
```

〔リスト20〕メイン関数その3

```
void main(int argc, char *argv[])
{
    /* ...... */
}
```

〔リスト21〕階乗計算(再帰呼び出し)

```
unsigned long fact(unsigned long ul)
{
    return ul == 0L ? 1L : ul * fact(ul - 1);
}
```

## 関数とマクロの扱いの違い

N先輩：次はマクロの話をしようか.マクロは,

```
#define name text
```

のように記述するもので,コンパイラじゃなくてプリプロセッサが処理するんだ.数行の簡単な関数をマクロにすると関数呼び出しのオーバヘッドがなくなるから,Cのライブラリ関数の中でもマクロで定義されているものがあるんだ.このマクロは関数と違って,副作用があるからライブラリ関数のどれがマクロか知っておく必要があるんだ(ANSI Cだとマクロに副作用がないことが保証されているが,コンパイラによってはANSI C完全準拠のコンパイルスイッチを指定しなければ副作用のあるマクロに展開するものもある).

●副作用1

引き数にインクリメント演算子(++),デクリメント演算子(--),関数呼び出しなどを使用しない.

```
#define max(a,b)
                (((a) > (b)) ? (a) : (b))
```

は最大値を返すマクロだけど,引き数が二度評価されるから,

```
c = max(a++, b++);
```

のように記述すると誤動作するんだ.

●副作用2

自作マクロでは引き数を()で囲む,

```
#define square(x) x*x
```

は square(x + 1) のように使うと誤動作するから,

```
#define square(x) ((x)*(x))
```

と記述しなくてはいけないんだ.

マクロはプログラム中だけ見ると関数と同じように見えるから,このような不具合があると,見落とす場合も少なくないんだ.でも,マクロは関数呼び出しのオーバヘッドがなくなって,しかも,引き数の型に依存しないから,うまく使えば非常に便利なものなんだ.

B君　：どういうことです.

N先輩：たとえば,さっきのマクロ max だと引き数が int,long,double などどれでも使用できるだろ.関数にすると引き数の型ごとに関数を作成する必要があるんだ.

B君　：あっ,そうですね.

〔リスト22〕16進表示その1(再帰呼び出し)

```
void puthexl(unsigned long ul)
{
    static const char xdigit[] = {"0123456789ABCDEF"};

    if (ul >= 0x10)
        puthexl(ul >> 4);
    putch(xdigit[(int)ul & 0x0f]);
}
```

〔表1〕実行時間(Pentiumプロセッサ150MHz)

| | VC++ 1.5 | VC++ 5.0 |
|---|---|---|
| リスト22(再帰呼び出し) | 44秒 | 12秒 |
| リスト23(ループ) | 51秒 | 19秒 |
| リスト24(ライブラリ) | 55秒 | 50秒 |

putch関数をダミー関数にして10,000,000回ループさせた

〔リスト23〕16進表示その2

```
    /*  符号無整数 → 16 進文字列   */

void puthexl(unsigned long ul)
{
    static const char xdigit[] = {"0123456789ABCDEF"};
    char buf[10];
    char *str1, *str2;

    str1 = str2 = buf;
    do {
        *str2++  = xdigit[(int)ul & 0x0f]; /* 余り Ascii 変換
*/
    } while ((ul >>= 4) > 0);
        *str2 = '¥0';
        while (str1 < --str2) { /*16 進文字列を逆順に並べ替え */
            char ch;

            ch  = *str1;
                *str1++ = *str2;
                *str2   = ch;
        }
    for (str1 = buf; *str1; str1++)
        putch(*str1);
```

〔リスト24〕16進表示その3

```
#include <stdlib.h>

void puthexl(unsigned long ul)
{
    char buf[10];
    char *str;

    ltoa(ul, buf, 16);
    for (str = buf; *str; str++)
        putch(*str);
}
```

## 再帰呼び出し(Recursive call)

### ●再帰呼び出しとは

N先輩：B君は再帰呼び出しはわかるかい.

B君　：いいえ.

N先輩：再帰呼び出しというのはひと言で言うと自分自身を呼び出すことなんだ. 自分自身を呼び出すといっても, 無制限に呼び出せばスタックがオーバフローしてしまうから, なんらかの抜け出し条件が必要になるんだけど. たとえば前述した階乗計算を再帰呼び出しで記述するとリスト21のようになるんだ. でも, 一般的に再帰呼び出しは自分自身を関数コールするオーバヘッドがあるから, 速度的には若干, 不利になるんだ. だから, 最後に自分自身を呼び出す場合は前述のリスト2のようにループに置き換える場合も少なくないんだ(リスト21のプログラムを関数の先頭へのジャンプに置き換えるコードを生成するコンパイラもあるようだ).

B君　：そうなんですか.

N先輩：再帰呼び出しは「Call by reference」では記述できなくて, かならず「Call by value」でないといけないんだ. 試しに, リスト21をアドレス渡しに書き変えてみてごらん. できないと思うけど.

B君　：???

### ●再帰呼び出しの利点

B君　：速度的に不利なら, 再帰呼び出しなんて意味ないですね.

N先輩：再帰呼び出しにも利点があるんだ. 再帰呼び出しの利点はアルゴリズムが簡単になる点なんだ. リスト21とリスト2だと簡単すぎてそれほど違いがわからないけど, もう一つ簡単な例題を示してみようか. リスト22とリスト23はどちらもunsigned longを16進表示するプログラムなんだ. リスト22だと再帰呼び出しを行うことで, 上位桁から1バイトずつ表示を行っているから, 非常に簡単になってるよね.

B君　：そうですね.

N先輩：リスト23だとループを使用して下位桁から変換しているから, 文字列を逆順に変換する操作なんかの余分な処理が増えてリストが少し読みにくくなってるよね. 16ビットのVC++ 1.5とintが32ビットのVC++ 5.0でputch関数をダミー関数,

```
void putch(int ch)
{
}
```

にして10,000,000回ループさせたときの実行時間を表1に示すよ. 表1には参考のためリスト24のようにライブラリのltoa関数を使用したときの時間も示しているけど. この表を見るとリスト23だといったんバッファに代入する処理や文字列を逆順にする処理がオーバヘッドになっていて, リスト22の再帰呼び出しによるオーバヘッドを少し上回っている感じだね. これらを見ると再帰呼び出しはオーバヘッドがあるけど, 再帰呼び出しの結果, アルゴリズムが簡単になれば再帰呼び出しのオーバヘッドを相殺することもある

〔リスト 25〕通信基本モジュール(FA 環境)

```
#define RSDATA      0   /* 8251A 送受信データ ポート オフセット */
#define RSSTAT      2   /* 8251A ステータス   ポート オフセット */
#define RSCMD       2   /* 8251A コマンド     ポート オフセット */
#define TxRDY       0x01    /* 送信レディ ビット                */
#define RxRDY       0x02    /* 受信レディ ビット                */
#define NOTRDY      -1
#define NOERR       0

int inp(unsigned int), outp(unsigned int, int);

static unsigned short port51[] = {  /* RS-232-C ポート アドレス  */
    0x30, 0x38
};

void init_aux(int chan)     /* RS-232-C イニシャライズ       */
{
    static struct {     /* 初期化データ             */
        unsigned char mode; /* 通信モード              */
        unsigned char cmd51;  /* 8251A コマンド・レジスタ 設定コマンド */
    } sccinit[] = {
        {
            0xce,   /*   モード・インストラクション
                         (2 ストップ ビット, パリティ無, 8 ビット, x16) */
            0x37    /* コマンド・インストラクション
                         (RTS, エラーリセット, RxE, DTR, TxEN)  */
        },
        {
            0xce,   /*   モード・インストラクション
                         (2 ストップ ビット, パリティ無, 8 ビット, x16) */
            0x37    /* コマンド・インストラクション
                         (RTS, エラーリセット, RxE, DTR, TxEN)  */
        }
    };
    unsigned short port;

    port = port51[chan];
    outp(port + RSCMD, 0);      /* ダミー インストラクション    */
    outp(port + RSCMD, 0);      /* ダミー インストラクション    */
    outp(port + RSCMD, 0);      /* ダミー インストラクション    */
    outp(port + RSCMD, 0x40);    /* 内部リセット              */
    outp(port + RSCMD, sccinit[chan].mode);
                                 /* モードパラメータ 初期設定   */
    outp(port + RSCMD, sccinit[chan].cmd51);
                                 /* コマンドレジスタ 初期設定   */
}

/*
    データ 1 バイト 受信処理

    返り値  0 ～ 0xff   受信データ
        NOTRDY      受信データ 無し
*/
int getc_aux(int chan)
{
    unsigned short port;

    port = port51[chan];
    return (inp(port + RSSTAT) & RxRDY) ?
        inp(port + RSDATA) : NOTRDY;
}

/*
    データ 1 バイト 送信処理

    返り値  NOERR       正常終了
        NOTRDY      送信ビジー
*/
int putc_aux(int chan, int data)
{
    unsigned short port;

    port = port51[chan];
    if (inp(port + RSSTAT) & TxRDY) {    /* TxRDY チェック ?    */
        outp(port + RSDATA, data);      /* データ 1 バイト 送信    */
        return NOERR;
    } else
        return NOTRDY;
}
```

〔リスト 26〕通信基本モジュール(MS-DOS 環境)

```
#include <stdio.h>
#include <process.h>
#include <dos.h>

#define NOTRDY      -1
#define NOERR       0

void init_aux(int chan)     /* RS-232-C イニシャライズ   */
{
#ifdef  IBMPC
    system("MODE COM1:9600,N,8,2");
#else
    system("SPEED R0 9600 PN B8 S2 NONE");
#endif
}

#ifndef __TURBOC__
int ioctl(int handle, int func, void *argdx, int argcx)
{
    union REGS iregs;

    iregs.h.ah = 0x44;
    iregs.h.al = (unsigned char)func;
                        /* IOCTL サブ・ファンクション   */
    iregs.x.bx = handle;    /* ファイルハンドル or ドライブ番号   */
    iregs.x.cx = argcx;
    iregs.x.dx = (unsigned)argdx;
    intdos(&iregs, &iregs);
    if (iregs.x.cflag)
        return -1;          /* エラー           */
    if (func == 0 || func == 1 || func == 9 || func == 10)
        return iregs.x.dx;
    return iregs.x.ax;
}
#endif

/*
    データ 1 バイト 受信処理

    返り値  0 ～ 0xff   受信データ
        NOTRDY      受信データ 無し
*/
int getc_aux(int chan)
{
    int stat;

    stat = ioctl(fileno(stdaux), 6, NULL, 0);
                                    /* 入力ステータス の取得      */
    if (stat != -1
      && (stat & 0xff) == 0xff)     /* 補助入力 レディ 状態 ?    */
        return bdos(0x03, 0, 0) & 0xff;
                                    /* 補助入力データ 表示       */
    else
        return NOTRDY;
}

/*
    データ 1 バイト 送信処理

    返り値  NOERR       正常終了
        NOTRDY      送信ビジー
*/
int putc_aux(int chan, int data)
{
    int stat;

    stat = ioctl(fileno(stdaux), 7, NULL, 0);
                                    /* 出力ステータスの取得 */
    if (stat != -1
      && (stat & 0xff) == 0xff) {   /* 補助出力 レディ 状態 ?    */
        bdos(0x04, data, 0);        /* データ 1 バイト 送信     */
        return NOERR;
    } else
        return NOTRDY;
}
```

注)ここで示した関数はIOCTL機能をサポートしていない.DOSでは誤動作する.

ことがわかるね．

## クロス環境の例

N先輩：うちは主としてFA関連の仕事をしているよね．
B君　：はい．
N先輩：FA関連だとある程度，規模が大きくなるとユニットごとにCPUを搭載して相互に通信を行いながら協調して動作するシステムも少なくないんだ．FA関連のシステムをパソコンで構成すればデバッグ環境には困らないんだけど，VMEバスのボードなんかを使用してシステムを構成するとデバッグ環境がいまひとつなんだ．Windows環境だとソースレベルの高度なデバッガが常識で，いろいろなデバッガがあるよね．
B君　：はい．
N先輩：だから僕は通信関係のデバッグはMS-DOS環境で行うようにしているんだ．通信の基本となるモジュールは
(1) 初期化
(2) 1バイト送信
(3) 1バイト受信
の三つだよね．
B君　：そうですね．
N先輩：ハードウェアに直接関係のある，これらのモジュールに関してFA環境とMS-DOS環境とで同じ入出力インターフェースの関数を作成しておくわけだ．リスト25はFA環境の例で簡単のため，非割り込み駆動にしているけど．FA環境では通常，割り込み駆動にするから，適切な例じゃないけど（I/Oのリカバリタイムも本質的ではないため，省略している）．リスト26はMS-DOS環境の例になるよ．リスト25だと複数のチャネルをサポートしてるけど，リスト26のパソコン版ではチャネルを無視して1チャネルしかサポートしてないけど．このように入出力インターフェースをあわせて作成しておいて別々のライブラリに登録しておけば，リンク時にライブラリを差し替えるだけで良くなるよね．
B君　：そうですね．
N先輩：別々の関数名で作成してコンパイル時に#defineで設定するという手も考えられるけどね．たとえば，

```
void init_aux_rom(int chan);
int getc_aux_rom(int chan);
int putc_aux_rom(int chan, int data);
void init_aux_dos(int chan);
int getc_aux_dos(int chan);
int putc_aux_dos(int chan, int data);
```

としておいて

```
#define init_aux    init_aux_rom
#define getc_aux    getc_aux_rom
#define putc_aux    putc_aux_rom
```

のようにすれば良いだろうね．このようにするとFA環境のプログラムをMS-DOS環境で1チャネルずつ，ある程度までデバッグできるようになるんだ．少なくとも，アルゴリズムの検証は行えるようになるから，その後，FA環境に移せばデバッグがかなり楽になるんだよ．
B君　：そうかもしれないですね．
N先輩：今日はこれくらいにしておこうか．
B君　：はい，ありがとうございました．

### おわりに

関数に関して，思い付くことを羅列したため，少しまとまりのないものになってしまいました．この原稿を書くにあたって，自分の場合を振り返ってみると，筆者が今まで作成してきた関数は，うまくない関数のほうが多かったように思います．そこで，自分自身の反省の意を込めてCプログラムの関数について改めて考察してみました．

今回，示した例も読者のみなさん（とくに中級以上の皆さん）にとっては，うまくない関数があるかもしれません．そのときは反面教師として，よりうまい方法を考えてみてください．

なかしま・のぶゆき　（株）Unix

第4章

**効率よくデバッグする方法を考える**

# デバッグの前準備と心得

中島 信行



デバッグというのはプログラム作成の最終段階での作業ですが，最初の設計時に手を抜いているとデバッグ途中で振り出しに戻ったり，プロジェクトが中止になったりといった最悪のケースも起こり得ます．デバッグ段階になってプログラム作成の半ばにさしかかったところと思ったほうが無難です．デバッグの仕方にもピンからキリまであり，またバグをゼロにすることは非常に難しいため，デバッグをいつ切り上げるかといった判断も重要です．

本章では主としてデバッガを使わない一般的なデバッグ論について紹介します．前章までと一部重複する部分も出てきますが，それだけ重要だということでお許しください．（筆者）

## デバッグの前に

B君　：先輩，おはようございます．

N先輩：おはよう．ところで，B君はデバッグはどうしてるんだい．

B君　：C++ Builderの統合環境でやってますけど．

N先輩：それ以外の方法は使ったことがあるかい．

B君　：Turbo Debuggerを使ったことがあります．

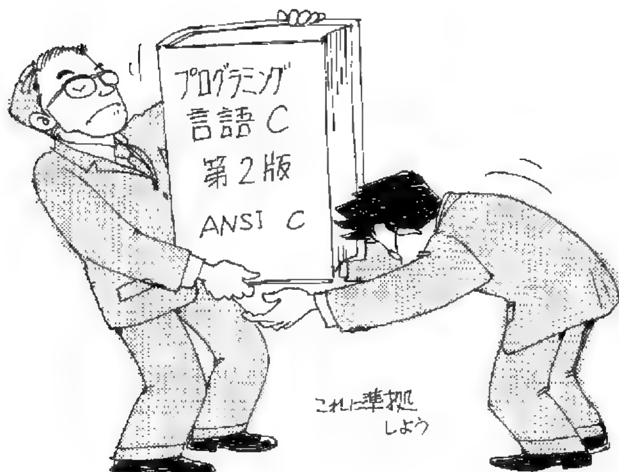
N先輩：それじゃ，今日はデバッグの四方山話（よもやまばなし）をしてみようか．デバッグでもっとも重要な点はなんだと思う．

B君　：そりゃ，バグをなくすことでしょう．

N先輩：そのとおり，当り前だね．そのためには，最初からバグのないコーディングをすれば良いんだけど，そんなことができるくらいなら誰も苦労しないよね．

B君　：そうですね．

N先輩：でもね，心がけ次第でバグを少なくすることはできるんだよ．まあ，最初にデバッグの前にバグを少なくするためのポイントをいくつか挙げてみよう．




### ● 自分なりのコーディングスタイルを決める

N先輩：B君はどんなコーディングスタイルを取ってるんだい．

B君　：諸先輩方のソースをつつくことが多いですから，そのソースの書き方を真似していますけど．

N先輩：コーディングスタイルはエディタや日本語IMEなんかのように一種の宗教みたいなところがあるんだけど，自分のコーディングスタイルと大きく異なるプログラムは読みにくいものなんだ．僕はその昔，プリティプリントというCプログラムの書式整形ユーティリティを自分なりのコーディングスタイルにあわせて改造していて，他人のプログラムを読まないといけない場合なんかには一度，このユーティリティに通してから読んでいくことも少なくないんだ．自分のコーディングスタイルを他人に強要することはできないんだけど，自分なりのスタイルを確立していくことは必要だと思うよ．たとえば，

（1）字下げは1TAB（4カラム）を単位とする

（2）単行演算子はスペースをあけない

`*  &  -  !  ~  ++  --` （型名）

（例）`if(!name)`

（3）二項演算子，代入演算子は前後にスペースを入れる

```
*   /   %   +   -   >>   <<   <   >   <=
>=  ==  !=  &   ^   |   &&   ||  ?:
=   +=  -=  *=  /=  %=  >>=  <<=
&=  ^=  |=
```

（例）`c_=_getchar();`（_はスペース）

（4）関数名と“(”の間にスペースを入れない

（例）`func(arg1, arg2);`

（5）配列名と“[”の間にスペースを入れない

（例）`array[10]`

（6）`if`，`switch`，`while`，`for`と“(”の間にスペー

スを入れる

（例） if_(flag)

（7）空文でも一行にする

（例） if_(flag)

```
;
```

（8）if～else if～elseのスタイル

```
if_(flag1)_{
    .....;
}_else_if_(flag2)_{
    .....;
}_else_{
    .....;
}
```

（9）whileのスタイル

```
while_(flag)_{
    .....;
}
```

（10）do～whileのスタイル

```
do_{
    .....;
}_while_(flag);
```

（11）forのスタイル

```
for_(式1;_式2;_式3)_{
    .....;
}
```

（12）switchのスタイル

```
switch_(式)_{
case(定数式1) :
    .....;
    break;
case(定数式2) :
    .....;
    break;
default :
    .....;
    break;
}
```

という感じかな．仕事だと課単位とか部単位でコーディングスタイルを決めておくことも重要だと思うよ．うちもそれなりの基準は決めてあるんだけど．僕の場合はCのバイブルといわれていた「プログラミング言語C」を元にして少しずつ変化してきて，今のコーディングスタイルになっちゃったけど，今，参考にするならANSI Cに対応した「プログラミング言語C第2版」あたりが良いかもしれないね．

● 変数名の決め方

N先輩：それに，変数名の決め方にしてもある程度約束事を決めておけばバグの減少につながるんだ．たとえば，

p――ポインタ
c――char変数
u――unsigned short変数
l――long変数
n――カウンタ
f――フラグ

といったように最初の一文字で変数のタイプを表すといった方法もあるけど，この辺の約束事は，いろいろな方法が考えられるから，単純に決められないんだけどね．

● プロトタイプ宣言を行ってコンパイラの警告レベルを上げる

N先輩：B君はC++ Builderを使ってるみたいだけど，コンパイルスイッチに-wをつけてるかい．

B君　：それって何ですか．

N先輩：プロジェクト(P)|オプション(O)のコンパイラを選択して警告の表示ですべて表示(A)をチェックするとコンパイルスイッチに-wが指定されるんだ．

B君　：それだったら，デフォルトのまま使ってますね．

N先輩：-wを指定しておいた上で，警告が表示されなくなるまでプログラムを修正するというのが，ミス防止の第一歩だと思うよ．

B君　：今後，気をつけます．

N先輩：B君はCのプログラムを書くこともあるのかい．

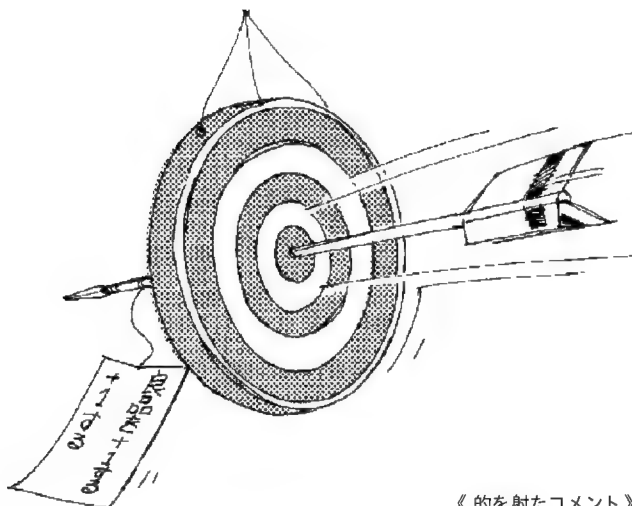
B君　：ありますよ．そのときは，BC++を使いますけどね．C++ Builderでもほとんど Cみたいなレベルでプログラムはかいてますけどね．

N先輩：C++のモードだと，プロトタイプ宣言が必要だから，問題ないけど，BC++で書くときはプロトタイプ宣言をきちんとしておくってのはやってるかい．

B君　：はい．一応やってます．

N先輩：プロトタイプ宣言をしておけばケアレスミスをコンパイラにチェックさせることができるから，この段階でバグが減らせるんだ（この段階のミスはバグというよりもコーディングミスが多いと思うが）．コンパイラの警告レベルはたいてい，数段階あって，警告レベルを上げてコンパイルすればケアレスミスの多くを見つけ出すことができるんだ．B君が使ってるBC++だと-w，MS-C，VC++だと/W4といったスイッチを指定してコンパイルすればケアレスミスの大半はなくせると思うよ．もっとも，警告レベルを上げると余計なお節介的な警告も増えてきて煩わしく感じることも少なくないんだけど，警告を極力消すようにコーディングしていけばケアレスミスは少なくなってくるはずだよ．

B君　：昔に書かれたANSI C以前のコンパイラ用のプログラムを保守しなきゃいけないことがあるんですけど，そ



《的を射たコメント》

んなときはどうするんですか．

N先輩：古いC（K&R C）の場合はANSI C用のプロトタイプ宣言とK&R Cのプロトタイプ宣言の両方を記述しておいて，エラーチェックだけANSI C準拠のコンパイラでやれば良いんだよ．もっとも，古いソースをつつくときはいちいち，ANSI C用のプロトタイプ宣言を追加していくのもめんどうだから，LINTでエラーチェックするという方法もあるんだ．

B君　：LINTって何ですか．

N先輩：LINTはプロトタイプ宣言がない時代のものだけど，ソースを見て，自動で内部的にプロトタイプ宣言みたいなものを生成してエラーチェックしてくれるエラーチェック専用のユーティリティなんだ．コンパイラだと複数のファイルに渡ったチェックはできないんだけど，LINTを使えば複数のファイルに渡ったチェックができるから，ファイル間でプロトタイプ宣言や外部参照の変数の宣言が違っているといったミスも見つけ出すことができるんだ．もっとも，複数のファイルに渡ってチェックを行うから時間がかかるという欠点をもってるんだけどね．

B君　：うちにはLINTはないんですか．

N先輩：K&R CレベルのLINTはあるんだけど，今は誰も使っていないと思うよ．ANSI C準拠のLINTだったら，使ってみても良いんだけど，もってないんだよ．最近は広告も見かけないしね．インターネットで探せば，あるみたいだけどね．

● 半年後の自分に説明するようにコメントを書く

N先輩：的を射たコメントはプログラムを読みやすくするんだけど，B君はコメントは書いてるかい．

B君　：一応書いてますけど．

N先輩：昔の自分のプログラムを眺めてみると，タコなプログラムを書いていたんだなと思ったり，こいつ，けっこう賢いじゃん，と思ったり，その時々の経験のレベルで，いろんな感じ方をするものなんだけど．

B君　：そういうもんですか．

N先輩：自分が書いたプログラムでも，時間がたてば，他人のプログラムとまでは言わないけど，客観的に見るようになるものなんだ．だから，コメントはとりあえず，自分のために書くことから始めたら良いんじゃないかと思うよ．C++ Builderなんかのメーリングリストを見ていると件名に「教えてください」とか，「初めまして」とか書いている人がいるけど，こんな件名だと，自分が後から見ても内容がわからないよね．

B君　：そうですね．

N先輩：それと似たことだけど，書いちゃいけないコメントってのがあるのを知ってるかい．

B君　：そんなのあるんですか．

N先輩：たとえばね，

```
var++;  /* +1 インクリメント */
```

というのは書いちゃいけないコメントなんだ．

B君　：どうしてです？

N先輩：初心者はうろ覚えの部分が多いから，こんなコメントを書くことがあるんだけど，Cをやってる人が見れば，

```
var++;
```

だけでvarを＋1インクリメントしてるのはわかるから，不要なんだよ．不要なコメントがあちこちにあると，肝心のコメントが目立たなくなってしまうから，プログラムの読みやすさを損ねるんだよ．これは，

```
goodcnt++;  /* 良品数を ＋1 する */
```

というような感じのコメントにすればまともなコメントになるんだけどね．

B君　：ああ，そうですね．

N先輩：初心者のころにはプログラムは他人がわかりやすいようにコメントを付けて書きなさいといったことを言われるもんだけど，他人にわかりやすいコメントというのはなかなかイメージしがたいよね．僕はとりあえず自分に説明するようなコメントから始めれば良いんじゃないかと思ってるよ．その自分も，半年くらい先のプログラムをかなり忘れてるはずの自分を想定して説明するように書けば良いんじゃないかな．これも抽象的でイメージしにくいかもしれないけど，困って人に話を聞いてもらってるときに自分で間違いに気づくってことがあるんだけど，人に説明しながらプログラムは書けないから，自分に説明するようなコメントを書くって感じかな．

B君　：僕も，人に説明してるときに，間違いに気づいたことはありますよ．

N先輩：誰にでもあると思うよ．後から読んでもわからないこ

とを書いても仕方がないから，数か月後に自分で読み返してもわかるように，自分に説明するように書くことを心がけていれば，だんだんとコメントの書き方がわかってくるんじゃないかと思うよ．

B君　：雑誌なんかに，コメントは英語で書けと載ってるものがありますけど，英語で書いたほうが良いんですかね．

N先輩：昔は日本語が表示できる環境が少なかったから，「コメントは英語で書け」といったことをいう人も多かったんだけど，日本人なんだから，コメントは日本語で書けば良いと思うよ．確かに，英語に慣れている上級者の英語のコメントは的を射てわかりやすいものなんだけど，初心者が書いた「スペルの間違った和製ローマ字英語のコメント」は読めたものじゃないよ．

B君　：なんか，ひどい目にあったことがあるみたいですね．

N先輩：昔は，担当者が辞めて保守できなくなったプログラムを保守してくれっていう，とんでもない依頼も何度かあったんだよ．たいていは断ったんだけど，いろいろなしがらみで断れないこともあったんだよ．

B君　：いろいろあるんですね．

N先輩：それと，仕様変更なんかがあって，後でプログラムを変更したときに，コメントもきっちり修正しておくのもけっこう大事なんだ．人のプログラムを読むときにコメントとプログラムが合ってなくて，困ったことがあったからね．

B君　：人のプログラムを改造するときによくわからなくてコメントを直さないってことがありましたけど．

N先輩：いろいろな人がプログラムを直しているとプログラミングのレベルも違うから，コメントが修正されないってこともあるだろうね．困ったことだけど．

B君　：そうですね．

## ● 他人のプログラムを読む

N先輩：B君は既存のプログラムの改造をしてるから，他人のプログラムを読む機会は多いと思うけど．

B君　：ええ，最近は人のプログラムを理解することから始めてますよ．

N先輩：他人のプログラムを読むという作業はけっこうたいへんなんだけど，学ぶ点も少なくないんだ．初心者のうちは影響を受けやすいから，へたなプログラムにはできるだけ接しないほうが良いんだけど，一流のプログラマが書いたプログラムというのは宝の山なんだ．プログラミングに限らず何をやるにしても最初は人真似から入るというのが常道だから，できるだけ多くの良質なプログラムに接するようにしたほうが良いと思うよ．今だとオープンソースのものがわりと簡単に手に入るから，興味がありそうなものを暇なときにじっくりと眺めてみるというのも良いかもしれないね．それに，フリーソフトウェアなどの他人の作ったユーティリティを元にして自分なりの拡張を行っていくという方法も有効かもしれないね．

〔リスト1〕間違って配列をポインタとして参照する

```
extern int *intvar;

void func(void)
{
    int n;

    for (n = 0; n < NINTVAR; n++) {
        intvar[n] = -1;
    }
}
```

〔リスト2〕ポインタ形式の引き数

```
void func1(char *pary)
{
}
```

〔リスト3〕配列形式の引き数

```
void func2(char pary[])
{
}
```

## ● ポインタと配列の違いを理解する

N先輩：B君はポインタと配列の違いはわかってるかい．

B君　：一応わかってるつもりです．

N先輩：ポインタと配列はトラブルメーカーでね，変数を定義しているファイルで，

```
int intvar[NINTVAR];
```

と記述して，この変数を参照している別のファイルでリスト1のようにポインタとしてコーディングされていると警告が出ずに正常にコンパイルされてしまうから，間違いになかなか気づかないこともあるんだ（LINTを使えばこんなバグでも発見できるんだけど）．Cではポインタと配列に関して，

```
array[0] と *array
```

のように同じようなスタイルのコーディングが許されてるから混乱しやすくなるんだ．配列は定数でポインタは変数だということが理解できればポインタと配列を間違えることは少なくなると思うんだけど，アセンブラを知らない人が理解するのは難しいかもしれないね（アセンブルリストを生成して眺めてみればポインタと配列の違いは一目瞭然なんだが）．

B君　：僕もアセンブラはわかりませんから，よく悩みますよ．

N先輩：ちょっと余談になるけど，リスト2とリスト3で関数func1とfunc2の引き数paryはどのような違いがあるかわかるかい．

B君　：だいぶ前に説明してもらったことがありますけど，どちらも同じことなんですよね．

N先輩：そうだね．関数の引き数だとchar *paryとchar pary[]には違いがないんだ．どっちもポインタとし



**News Flash ── VxWORKS搭載のNASAの火星探査機が火星着陸成功**
米WindRiver社のOSであるVxWORKSを搭載したNASAの火星探査ローバーSpiritが米国太平洋時間1月3日午後8時35分（日本時間4日午後1時35分）に火星着陸に成功した．Spiritのソフトウェア部分ははTORNADO開発環境およびデバッガにて開発された．

てまったく同じコードが生成されるんだ．これを初心者に説明するのは難しいんだけど，僕は次のように説明してお茶を濁してるんだよ．Cの関数だと引き数として定数をもつことができないから，関数の引き数はかならず変数になるんだ．だから，引き数を配列のスタイルで宣言してもポインタ(変数)とみなされることになるんだってね．

### ● プログラムのインデントに注意

N先輩：インデントは構造化プログラミングには不可欠で，プログラムを読みやすくするのに役立つんだけど，Cコンパイラにとってはインデントはまったく関係ないものなんだ．だから，リスト4のようなプログラムは意図したものと違った動作となるんだよ．最初から，リスト4のようなコーディングをすることはないと思うけど，ネストの深いif文に後から追加した場合などにはやっちゃうことがあるものなんだ．だから，リスト5のようにif文には{}を必ず付けるという癖を付けるのも良いかもしれないよ．

B君　：はい．

### ● 行を分ける

N先輩：Cだとカンマ(,)演算子を使って一行に多くの式がかけたりするよね．

B君　：はい．

N先輩：それに，代入の結果が値をもつから，リスト6をリスト7のように書いてしまうことが少なくないよね．

B君　：そうですね．

N先輩：僕も良くするから，リスト7のような書き方が悪いというわけじゃないけど，ソースレベルのデバッグという観点から見るとリスト6のように極力，行を分けて別の文にする書き方のほうがブレークポイントが別々に設定できるから便利なんだ．アセンブラ＋Cソースのモードで追いかけるときもリスト6のスタイルのほうが追いかけやすい気がするよ．それに，複雑な式になるとリスト7のような書き方だと一目で理解できなくなることも多いからね．最終的に最適化のスイッチを指定すれば最近のコンパイラは最適化が進んでるから，リスト6とリスト7だと生成コードに差はないと思うよ．

### ● 関数の入出力仕様を書く

N先輩：B君は関数の始めに何を書いてるの．

B君　：とくに書いてないですけど．

N先輩：関数の記述に先だって関数の入出力仕様をコメントで書いておくというのは重要なことなんだ(リスト8)．Cのプログラムは関数を単位として構成されるから，最初に入出力仕様が書いてあれば，関数を記述するときにも明確になるし，その関数を呼び出す箇所を記述するときに引き数の仕様を確認するときにも間違いが少なくなるんだ．入出力仕様は関数の引き数と返り値は当然のことだけど，外部変数で受け渡しを行っている場合はその外部変数の仕様も書いておく必要があるだろうね．

### ● プログラムは集中して書こう

N先輩：余談だけど，同じ仕様のプログラムをある人が一週間で作って，別の人が一か月で作ったとするとどちらが良いものができると思う？

B君　：それは一か月かけたほうが良いものができるんじゃな

〔リスト4〕間違ったインデント

```
    if (flag1)
        data1 = 1;
        data2 = 1;
    if (flag2)
        data2 = 0;
```

〔リスト5〕if文には{}を必ず付ける

```
    if (flag1) {
        data1 = 1;
        data2 = 1;
    }
    if (flag2) {
        data2 = 0;
    }
```

〔リスト6〕行を分けた後

```
    fp = fopen(name, "r");
    if (fp == NULL) {
        fprintf(stderr, "File %s not found.¥n", name);
        exit(1);
    }
```

〔リスト7〕行を分ける前

```
    if ((fp = fopen(name, "r")) == NULL) {
        fprintf(stderr, "File %s not found.¥n", name);
        exit(1);
    }
```

〔リスト8〕関数の入出力仕様をコメントで書いておく(ltona.c)

```
/*
    long→base進n桁文字列変換

    Bin. → n桁文字列 + '¥0'
            str[] サイズ >= n+1

    123  → "  123"
    0    → "    0"

    引き数  long lval;   変換数値
            char *str;   base進文字列格納アドレス (MSB→LSB : サイズ >= n+1)
            int     n;   変換桁数
            int  base;   変換基数 (±2～36, 負 : 符号付き, 正 : 符号なし)

    返り値 str
*/
char *ltona(long lval, char *str, int n, int base)
{
    /* 省略 */
}
```

いですか．

N先輩：何事にも能力差というものはあるけど，プログラミングというのは個人差の非常に大きいものの一つなんだ．プログラミングで飯を食っている人の間で比較してもトップクラスの人とボトムクラスの人だと軽く二桁以上の能力差はあるものなんだ．だから，プログラミングは時間をかければ良いものができるというものじゃなくて，逆に時間をかけるほど保守しにくいできの悪いプログラムができるもんなんだよ．

B君　：そう言われても，能力ってのは急に向上しないですしね．

N先輩：そのとおりだね，他人と比較しても始まらないよね．ところで，B君はプログラムを書いてるときに電話がかかってきて，電話の後でやってたことを忘れるって経験はあるかい．

B君　：時々ありますよ．

N先輩：同じ人でもそのときの体調や気分なんかでプログラムの出来が違ってくることがあるんだ．できるだけ良いプログラムを書こうと思ったら，プログラミングに集中するのが良いんだ．そのためには，電話なんかで中断されない環境を作ることがベストなんだけど，なかなか難しいかもしれないね．でも，僕は納期のないプロジェクトに駆り出されて数か月こもってプログラムを組んだことが何度かあるけど，電話がかからない環境というのは非常に仕事がはかどるものなんだ．だから，電話を取らない時間帯を作るってのも，もし，可能なら有効かもしれないね．

● 怠惰になろう

N先輩：B君はけっこうまめそうだね．

B君　：そうでもないですよ．

N先輩：プログラムというのは基本的に誰かが楽をするためにつくるものだけど，楽をしたいというのは誰でも思うものだよね．

B君　：はい．

N先輩：プログラムを使う人だけに楽をさせるのは不公平だから，プログラムを書く側もプログラミングに当たってできるだけ良い意味での手抜きをするべきだと思うよ．たとえば，

- この繰り返しのパターンはループにできないか
- この繰り返しのパターンは関数に（サブルーチン化）できないか
- これらの変数を集めて配列にすればループにできないか
- これらの変数を集めて構造体にすれば代入が一行にできるのではないか
- もっと簡単なアルゴリズムはないか
- この部分はどこかにあるプログラムをもってこれないか




なんていろいろと考えられるよね．プログラムをループやサブルーチン化によって短くすればデバッグする箇所も少なくなるから，デバッグも楽になるものなんだ．

## デバッグの心得

● 机上デバッグ

N先輩：B君は机上デバッグってのを知ってるかい．

B君　：いいえ．

N先輩：最近はソースレベルのデバッガが当り前になってきたから，プログラムを書いてコンパイルができるようになったら，すぐにデバッグにかかるといったことが多くなったけど．その昔，コンピュータが自由に使えなかった時代にはリストをプリントアウトして何度も読み返してこれでだいじょうぶだという確信をもってからコンピュータにかけたものなんだ．それでもなかなか思うように動いてくれなくて苦労したものだけど．プログラムがそれほど大きくなければソースレベルのデバッガでいきなりデバッグを始めるというのも有効な手段かもしれないけど，プログラムが大きい場合は，プログラムの全体像を頭の中にたたき込むために，プログラムを何度か読み返してみるということも必要になってくるものなんだ．「読書百遍意おのずから通ず」ということわざもあるしね．頭の中で動きをシミュレートしていくと間違いやむだな部分が見えてくることも少なくないよ．

● デバッグする前にケースを洗い出せ!!

N先輩：B君はデバッグの前にデバッグリストみたいなものは書いているかい．

B君　：いいえ．とりあえず動かしていろいろ操作してデバッグしてますけど．

N先輩：デバッグというのはすべてのケースを調べようとすると膨大な時間がかかるけど，デバッグする前には一応，デバッグ項目をすべて書き出しておいて，一つずつ潰していくことが必要なんだ．デバッグの途中でソースを書き直すとすでにデバッグ済みの箇所に影響が出る場合もあるから，本当は最終的にもう一度ひととおりデバッグする必要があるんだけど．Windows環境ならテストツールを使えば手順をスクリプトで記述して自動テストができるから，何度でも繰り返すのに楽だと思うけど．

〔リスト9〕assertマクロの例(VC++)

```
#ifdef NDEBUG

#define assert(exp) ((void)0)

#else

void _assert(void *, void *, unsigned);
#define assert(exp) ¥
    ((exp) ? (void) 0 : _assert(#exp, __FILE__, __LINE__))

#endif /* NDEBUG */
```

〔リスト10〕assertマクロの活用

```
/*

コンパイル・スイッチ

Visual C++ 5.0

cl /J /W4 /Zp assert.c

Microsoft (R) C Compiler  Version 5.10 / 6.00A /
7.0A, VC++ 1.0 / 1.5

cl /J /W4 /Zp assert.c /link /st:0x2800/cp:0x1000

C++ Builder

bcc32 -w assert.c

Borland C++  Version 3.00

bcc -w assert.c

Turbo C  Version 2.0 / Turbo C++  Version 1.0

tcc -w assert.c

LSI C-86 Ver 3.30

lcc assert.c

*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void func(void *pnt);

int main(void)
{
    func(NULL);
    return 0;
}

void func(void *pnt)
{
    assert(pnt != NULL);     /* ポインタが NULL の場合に
メッセージを表示して終了する. */
}
```

● デバッグに行き詰まったら気分転換をしよう!!

N先輩：B君はデバッグ中に行き詰まったことはあるかい．

B君　：何度もありますよ!!

N先輩：デバッグをしてるとひとつの不具合につまずいてなかなか次に進めないことがあるものなんだ．こういうときには自分自身が迷路に入り込んでしまっていることが少なくないから，いったん，迷路の外に出ることが必要なんだ．そんなときにはその不具合をほっておいて，次の項目をデバッグするとか，いっそのことまったく別の仕事をするといったことで，ふと解決の糸口が見えてくることがあるんだ．僕なんか，寝起きのときにバグの原因が浮かんでくることがよくあるんだけど，集中して考えていたときに短絡していた頭の中の回路が気分転換によって復旧されるといった感じなのかもしれないね．そばに聞き上手の人がいれば捕まえて現象を説明してみるというのも有効な手段なんだ．言葉にして他人に説明してみるという過程で迷路にはまり込んでいた自分の頭の中が整理されてバグの箇所が浮び上がってくるといったことも少なくないんだよ．

## printfデバッグの手法

N先輩：B君はprintfデバッグっていうのを知ってるかい．

B君　：いいえ．

N先輩：printf文を要所に埋め込んで変数の値などを表示させるデバッグ法をprintfデバッグというんだ．デバッグの基本だよ．僕が25年以上前に大型機を使用していたころはこれが唯一のデバッグ方法だったんだ．ソースレベルのデバッガが使用できればprintfデバッグは必要ないけど，ソースレベルのデバッガが使用できない環境というのは組み込み機器の場合は今でもあるから，最後の手段としてprintfデバッグを使うことがあるんだ．printfデバッグの一種としてANSI Cではassertマクロが定義されてるけど，assertマクロは知ってるかい．

B君　：いいえ．

● assertマクロの活用

N先輩：Cにはデバッグ用の機能としてassert関数が用意されてるんだ．assert関数はリスト9のようにマクロとして記述されていて，

```
void assert (int expression) ;
```

という形式になってるんだ．expressionが偽(0)のときに診断メッセージが出力されて，abort関数でプログラムを終了させるんだ．NULLポインタの検出

〔図1〕リスト10の実行結果

```
E:¥>assert
Assertion failed: pnt != NULL, file assert.c, line 45

abnormal program termination

E:¥>
```

(a) Visual C++ 5.0

```
E:¥>assert
Assertion failed: pnt != NULL, file assert.c, line 45

abnormal program termination

E:¥>
```

(b) Visual C++ 1.5

```
E:¥>assert

Assertion failed: pnt != NULL, file assert.c, line 45

Abnormal program termination

E:¥>
```

(c) C++ Builder 6

```
E:¥>assert
Assertion failed: pnt != NULL, file assert.c, line 45
Abnormal program termination

E:¥>
```

(d) Borland C++ 3.00

```
E:¥>assert
Assertion failed: pnt != (0), file assert.c, line 45

Abnormal program termination

E:¥>
```

(e) LSI C-86 3.30

などプログラムの論理的なエラーを検出するために使用するんだ．プログラム例をリスト10に，リスト10の実行結果を図1に示すよ．

B君　：assertマクロで記述した部分は最終的にどうするんですか．

N先輩：assertマクロはコンパイル時に#define NDEBUG（またはコンパイルスイッチ-DNDEBUG）を指定すれば何もしないマクロに置き変わるんだ．だから，そのままにしておいても，むだなコードは生成されないんだ．

### ● printfデバッグの実例

N先輩：ついでにいうとprintf文によるデバッグもassertマクロのスタイルにあわせておけば良いと思うよ．コンパイル時に，

```
#define NDEBUG
```

を定義すればコードが生成されないようにしておくんだ．たとえば，リスト11のようなデバッグ用のマクロを定義しておいて，知りたい変数の箇所で，

```
printl(ldata);
```

というコードを記述しておくんだ．この箇所が実行されると，

```
test.c(52) : idata = 12345 (3039)
test.c(53) : ldata = 12345 (3039)
```

〔リスト11〕printfデバッグ マクロその1

```
/*

コンパイル・スイッチ

Visual C++ 5.0

cl /J /W4 /Zp test.c

Microsoft (R) C Compiler  Version 5.10 / 6.00A /
7.0A, VC++ 1.0 / 1.5

cl /J /W4 /Zp test.c /link /st:0x2800/cp:0x1000

C++ Builder

bcc32 -w test.c

Borland C++  Version 3.00

bcc -w test.c

Turbo C  Version 2.0 / Turbo C++  Version 1.0

tcc -w test.c

LSI C-86 Ver 3.30

lcc test.c

*/

#include <stdio.h>

#ifdef NDEBUG

#define printd(exp) ((void)0)
#define printl(exp) ((void)0)

#else

#define printd(exp) printf("%s(%u) : %s = %d (%x)¥n",¥
            __FILE__, __LINE__, #exp, (int)(exp), (int)(exp))
#define printl(exp) printf("%s(%u) : %s = %ld (%lx)¥n",¥
            __FILE__, __LINE__, #exp, (long)(exp), (long)(exp))

#endif /* NDEBUG */

int  idata = 12345;
long ldata = 12345L;

int main(void)
{
    printd(idata);
    printl(ldata);
    return 0;
}
```

〔リスト 12〕printfデバッグマクロその2

```
/*

コンパイル・スイッチ

Visual C++ 5.0

cl /J /W4 /Zp test.c

Microsoft (R) C Compiler  Version 5.10 / 6.00A /
7.0A, VC++ 1.0 / 1.5

cl /J /W4 /Zp test.c /link /st:0x2800/cp:0x1000

C++ Builder

bcc32 -w test.c

Borland C++  Version 3.00

bcc -w test.c

Turbo C  Version 2.0 / Turbo C++  Version 1.0

tcc -w test.c

LSI C-86 Ver 3.30

lcc test.c

*/

#include <stdio.h>
#include <ctype.h>

#ifdef NDEBUG

#define printd(exp) ((void)0)
#define printl(exp) ((void)0)

#else

extern int fdebug;
#define printd(exp) printf(fdebug ? "%s(%u) : %s
= %d (%x)¥n" : "",¥
            __FILE__, __LINE__, #exp, (int)(exp), (int)(exp))
#define printl(exp) printf(fdebug ? "%s(%u) : %s
= %ld (%lx)¥n" : "",¥
            __FILE__, __LINE__, #exp, (long)(exp), (long)(exp))

#endif /* NDEBUG */

void getargs(int, char *[]);

int  idata = 12345;
long ldata = 12345L;
int  fdebug;

int main(int argc, char *argv[])
{
    getargs(argc, argv);              /* 引数の取得       */
    printd(idata);
    printl(ldata);
    return 0;
}

void getargs(int argc, char *argv[])  /* 引数の取得       */
{
    while (--argc) {
        char *p = *++argv;

        if (*p == '-') {
            while (*++p != '¥0') {
                switch (toupper(*p)) {
                case 'D':
                    fdebug = 1;
                    break;
                default :
                /*  usage();    */
                    break;
                }
            }
        }
    }
}
```

という形式で標準出力に出力されるから，この出力をファイルにリダイレクトすればエディタのタグジャンプ機能でソースをオープンして結果を該当箇所と見くらべられるから便利なんだ．

B君　：今後の参考にします．

N先輩：もっとも，実際にはデバッグ用のコードを埋め込んでおいて不具合が発生したときにすぐに見れるようにしておきたい場合も少なくないんだ．そんな場合には，

(1) コマンドラインのオプションにデバッグ用のコードを有効にするための隠しスイッチを設けておく
(2) 特定のキー操作でデバッグ用のコードを有効にできるようにする

というような方法を使うんだ．たとえば，リスト12のようなマクロを定義しておいて，コマンドラインのオプションでfdebugをセットできるようにしておけば良いわけだよ．

● OSの機能を使用しないprintf関数

N先輩：OSの機能が使用できるプログラム部分のデバッグならprintf関数で必要とする変数の値などを表示させることができるけど，

- 割り込み処理プログラム内
- 常駐プログラムの常駐部
- デバイスドライバ

なんかだと通常の手段ではOSの機能が使用できないから，printf関数が使用できないことがあるんだ．これらの部分はCodeViewなんかの通常のソースレベルデバッガでもうまくデバッグできないんだ(もっとも，一部の特殊なデバッガだと常駐プログラムの常駐部やデバイスドライバなどがデバッグできるモードがあるが)．OSの機能が使用できない箇所だと，

- テキストVRAMに直接書き込む
- ブザーを鳴らしてその箇所まで実行したことを知らせる
- プリンタポートを直接たたいてその箇所まで実行したことを知らせる

などの対策が必要となるんだ．プリンタポートを直接たたく場合は，ドットプリンタはキャリッジリターン(0Dh)やラインフィード(0Ah)を受け取るまでバッ

ファリングしてるから，これらを最後に付加して出力するようにしないといけないんだ．組み込み機器だと基板にLEDなんかを付けて，ソフトの立ち上げ時にどこまで実行しているのかを確認するために使うことがあるんだ．

B君　：そうですか．

N先輩：ハードウェア割り込み処理のデバッグの場合は，最初にハードウェア割り込みをソフトウェア割り込みとして呼び出してデバッグするというのも有効な方法なんだ．タイミングが問題になるような部分はデバッグできないけど，割り込み処理のおおよその流れのデバッグはできるから，単純なミスは発見できるんだよ．ハードウェア割り込み処理を間違えてると最悪の場合，そこでハングアップしてしまうから，単純なミスでも発見が難しいものなんだ．ソフトウェア割り込みとしてデバッグしてハングアップしないレベルにしておくだけでも，後のデバッグが楽になるものなんだよ．

## リモートデバッグの手法

N先輩：B君はリモートデバッグは知ってるよね．

B君　：はい．

N先輩：リモートデバッグはRS-232-Cなどを介したデバッグで，ターゲットの他にパソコンなどのディスクをもったホストが必要となるけど，けっこう有効な手段なんだ．リモートデバッグの利点はRS-232-C以外はターゲットがすべてのハードウェアを使用できることなんだ．

(1) グラフィックを使用したプログラムでもデバッガによって画面が制限を受けない
(2) デバッガが使用するメモリによってアプリケーションがメモリ不足になる可能性が少ない
(3) 割り込み処理の部分，常駐プログラムの常駐部などのデバッグも可能
(4) ターゲットがROM化環境でもデバッグできる

といった利点があるんだ．さっき言ったprintfデバッグだとプログラム内にデバッグ用のコードが散らばってしまうから，プログラムが多少見にくくなるけど，リモートデバッガのターゲット部分は別タスクとして記述するから，プログラム本体が見にくくなることはないんだ．うちではROM化環境の86系と68K系のリモートデバッガを作成していて，VMEバスのボードを使用したFA関連の仕事ではROM化するターゲットにはリモートデバッガを組み込んで出荷してるから，その辺のことはわかるよね．

B君　：はい．

N先輩：こうしておけば社内でのデバッグに活用できるのはもちろんのこと，現地でもノートパソコンを1台持っていけばプログラムの不具合をかなりのところまで追いかけることができるんだ．

## 入出力のデバッグの手法

N先輩：入出力のデバッグはハードウェアが絡んでくるから，けっこうめんどうなものなんだ．基本的にはprintfデバッグで対応可能だけど，組み込み機器の場合は，その前に，まずは入出力用の簡単なテストプログラムを書いてハードウェアが正常に動作することを確認しておく必要があるんだ．ハードウェアを操作するBIOSを作成するときにはBIOSのデバッグをするときにハードウェアの確認も行うことになるけど，このBIOSにデバッグ用の機能を埋め込んでおくという方法もあるんだ．たとえば，RS-232-CのBIOSであれば送信と受信のラインモニタ用のリングバッファを設けておいて送受信を行うごとに送受信データをセットしておくんだ．アプリケーションでこのリングバッファの内容を表示するモードを用意しておけば送受信データがモニタできるようになるんだよ．繰り返しテストをする場合には，入力をファイルから読み込めるようにしておけば便利だよ．出力結果もファイルに書き込むようにすれば，いくつかのパターンをバッチファイルで自動実行させて後からじっくりと追いかけることができるようになるんだ．

## リングバッファの管理法

### ● リングバッファとは

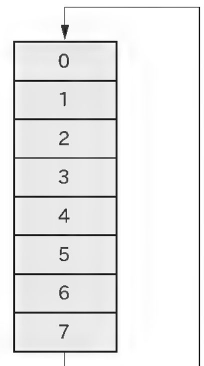
B君　：リングバッファってなんですか．

N先輩：リングバッファというのは，リングになったバッファのことなんだよ．

B君　：それって説明になってませんよ．

N先輩：そうだね．バッファサイズが大きいと説明しにくいから図2のように8個で考えると最初の1個目を0の位

〔図2〕リングバッファ



置，2個目を1の位置ってぐあいにセットしていくとするだろ．

B君　：そうすると，8個でいっぱいになりますよね．

N先輩：そうだね．それじゃ，9個目はどの位置にセットすればいいかわかるかい．

B君　：いっぱいだからセットするとこはないのでは？

N先輩：いっぱいだったらそうだけど，0の位置がすでに読まれてるとすれば9個目は0の位置にセットすればいいだろ．

B君　：そうですね．

N先輩：バッファの開始点と終了点を結んだ形にして結果的にリング状にしたバッファをリングバッファというんだよ．つまり，バッファの最後に書き込んだら，次はバッファの先頭に書き込むようにすればリングバッファになるんだ．

B君　：0の位置がまだ読まれてなかったらどうなるんですか．

N先輩：それはエラーになるんだよ．通信チップでオーバランエラーってのを知ってるかい．

B君　：いいえ．

N先輩：通信チップは受信バッファを数個もってるんだけど，受信バッファがいっぱいの状態でCPUが受信データを読まないうちに次のデータがきたら，データがオーバライト(上書き)されてオーバランエラーになるんだ．それと同じようなことだよ．

B君　：はあ～!?　ちょっと，ピンとこないですけど，リングバッファの管理はどんなふうにするんですか．

N先輩：リングバッファの管理は一般的には二つの方法があるんだ．僕が良く使うのは，

- 書き込みポインタ
- 読み込みポインタ
- カウンタ

の三つを使う方法だよ．

B君　：詳しく説明してくださいよ．

N先輩：書き込みポインタはデータを次に書き込む位置を指し示していて，読み込みポインタはデータを次に読み取る位置を指し示しているんだ．それと，カウンタはバッファ内のデータ数を示しているんだ．バッファに空きがあってデータをバッファにセットするときには，

```
書き込みポインタが指す位置にデータをセット;
++書き込みポインタ;
++カウンタ;
if(書き込みポインタがバッファの最後を越えた?)
   書き込みポインタ = 0(バッファ先頭);
```

といった手順になるんだよ．

B君　：なんとなくわかります．

N先輩：読み取りの場合は，

```
読み込みポインタが指す位置のデータを読む;
++読み込みポインタ;
--カウンタ;
if(読み込みポインタがバッファの最後を越えた?)
   読み込みポインタ = 0(バッファ先頭);
```

といった手順になるんだ．

B君　：書き込みと読み込みが追いかけっこしてる感じですね．

N先輩：そんな感じかな．バッファがいっぱいになる前に読みとっていけば書き込みと読み込みの追いかけっこが無限に続くイメージになるだろ．

B君　：そうですね．

N先輩：これがリングバッファのメリットなんだ．バッファの大きさが緩衝材になって，書き込みと読み込みの速度差をある程度吸収してくれることになるんだ．だから，通信BIOSなんかで受信割り込み処理で受信データをリングバッファにセットしていけばアプリケーション側はリングバッファがいっぱいになるまでに受信データを読み出せば良いからプログラムが楽になるんだ．

B君　：へえ～，そうなんですか．

N先輩：リングバッファの最後の判定は書き込み/読み込みポインタをバッファサイズと比較するという方法もあるんだけど，バッファサイズを2の累乗にしてバッファサイズ－1とのANDを取るという方法もあるんだ．たとえば，図2のようにバッファサイズが8バイトだったら書き込み/読み込みポインタ &(8－1)とするんだ．こうすれば0～7をリング状に繰り返すことになって，バッファの最後と比較する必要がなくなるからメリットがあるんだ．この方法だと書き込みは，

```
書き込みポインタが指す位置にデータをセット;
書き込みポインタ = 書き込みポインタ &
                        (バッファサイズ-1);
++カウンタ;
```

というような感じになって，さっきよりも少しシンプルだろ．

B君　：あっ，そうですね．

N先輩：でもね，この方法はバッファサイズが2の累乗でしか調整できないという欠点もあるから，RAMが少ないターゲットだとバッファ効率上使えないこともあるんだけど．

B君　：そうですね．16が32になる程度なら良いでしょうけど，16384が32768くらいになるとちょっとつらくなるかもしれませんね．

### ● 書き込みポインタと読み出しポインタを使う方法

N先輩：リングバッファの管理方法には，もう一つ書き込みポインタと読み込みポインタだけを使用してカウンタを用いない方法もあるんだ．

B君　：バッファ内のデータ数はどうやって判定するんですか．

N先輩：バッファ内のデータ数は書き込みポインタと読み込みポインタの差で判断するんだ．書き込みポインタと読

み込みポインタが等しいときはバッファが空の状態で，バッファがいっぱいかどうかは書き込みポインタ＋1（当然バッファの最後を越えたら先頭に戻す）が読み込みポインタと同じになるかどうかで判断するんだ．

B君　：だったら，バッファサイズいっぱいに書き込むと書き込みポインタと読み込みポインタが等しくなりますよね．

N先輩：そうだね．だから，空の状態と区別できなくなるからバッファリングできるデータ数がバッファサイズ－1となるんだ．

B君　：どっちの方法がいいんですかね．

N先輩：どっちの方式が良いとか悪いとかは難しいけど，自分の頭で考えやすい方法をいつも使うようにしたほうが良いと思うよ．両方を併用していると頭がこんがらがってくるはずだからね．僕は頭が慣れているから，いつも最初の方式でリングバッファを管理しているよ．最初の方法のメリットはね，バッファサイズをもってるから，バッファサイズの1/3とか，1/4とかが簡単に判断できるところなんだ．

B君　：それって，なんのために必要なんですか．

N先輩：通信では受信バッファが3/4以上になったら，相手の送信を止める処理を行って，1/4以下になったら，相手の送信を再開させるって感じのフロー制御を行うことがあるんだ．3/4，1/4は2/3，1/3とか好みによって変わるんだけど．こんなときはバッファサイズをもってるほうがプログラムがすっきりすると思うよ．

B君　：そうみたいですね．それじゃ，僕も最初の方式を使うようにしてみます．

## 移動平均
――― 入力がバラつくときの対処法

### ● A-D変換のデータを処理するとき

B君　：そういえば，今，A-D変換入力を扱うプログラムを作っているんですけど，同じ位置でも入力がバラつくんですよね．なんか良い方法はないですか．

N先輩：アナログ入力を扱う場合はノイズ対策も兼ねて平均を取った値を使うのが普通なんだよ．

B君　：どういうことですか．

N先輩：アナログ入力は同じ入力をしているようでもA-D変換を行った値は少しふらついてるものなんだよ．とくに，ノイズが多い環境ではふらつく範囲も大きくなるものなんだ．こういうときには，5回とか，10回とか入力を読んでその平均値を使えばある程度のふらつきは抑えられるんだ．

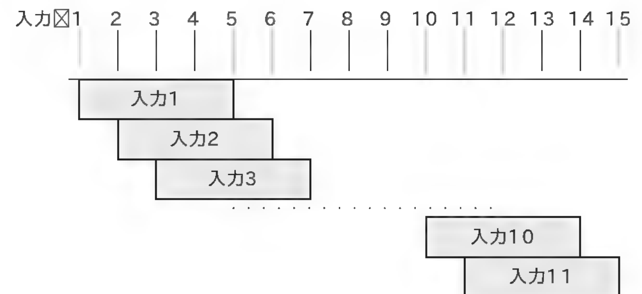
B君　：5回も10回もリードすると，A-D変換にかなりの時間がかかってしまうんじゃないですか．

N先輩：入力は1サイクルに1回行うだけだから，平均しない場合と差はないんだよ．

〔図3〕5回の平均




〔図4〕5回の移動平均




B君　：でも，それだと図3のようになって，入力値を使うサイクルが長くなってしまわないんですか．

N先輩：図3のような平均の取り方をする場合もなくはないと思うけど，普通は移動平均を取るんだ．

B君　：移動平均ってなんですか．

N先輩：図4のような方法だよ．たとえば，5回の平均を取るとすれば5個ぶんが溜るまでは制御を行わないで，5個溜ったら，最初の平均を取ってそれを入力値として使うんだ．それで，次の入力値を読んだときはいちばん古い入力値を捨てて残りの5個分の平均を取るんだ．

B君　：あっ，そうか．そうすれば5個溜ってからは毎サイクル入力値があることになりますね．

### ● 平均を取るということ

N先輩：そういうことだね．それと，平均の取り方にもいろいろと方法があってね．

B君　：どんな方法があるんですか．

N先輩：いちばん単純なのは5個の合計を5で割った値を使う方法なんだけど．

B君　：それって当たり前ですよね．

N先輩：ノイズ環境が良ければそれでまったく問題ないんだけど，ノイズ対策として5個の内の最大と最小を除いて残り3個の合計を3で割るって方法もあるんだ．

B君　：なんか，オリンピックの体操の採点法みたいですね．

N先輩：その方法の延長線上として最初5個の平均を取ってその平均より，ある％以上大きいか小さい値を捨てる方法とか，そのアプリケーションに応じていろいろとくふうする場合もあるんだよ．

B君　：最後の方法はなんかややこしそうですね．それで，平均を取る個数は普通何個なんですか．

N先輩：まあ，平均を取るってことは入力の変化を鈍くするこ

とだから，あまり数を多くすると応答が悪くなってしまうので，ほどほどの個数にしないといけないんだ．たとえば，個数を100個にすると100個目が2倍になっても1%しか効いてこないだろ．

B君　：そうですね．

N先輩：入力周期にもよるけど，たいてい10個以下じゃないかと思うよ．僕がいちばん手抜きをしたときは3個のうちの最大と最小を捨てるって方法を取ったときだと思うけど．

B君　：その方法だと残りが1個になって平均を取らなくていいから，確かに手抜きの方法ですね．

N先輩：それは8ビットのZ80の時代にやったことのある方法なんだけど割算命令がないから，平均を取らないってのもそれなりにメリットがあったんだ．

B君　：割算を速くするんだったら，割算がビットシフトになるように6個のうちの最大と最小を捨てて4個の平均を取るとか，4個のうちの最大と最小を捨てて2個の平均を取るって方法もありますね．

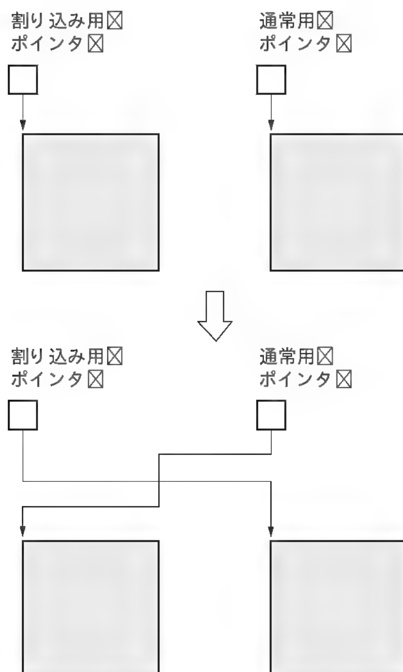
N先輩：だんだんわかってきたじゃない．それと，A-D変換に限れば下位ビットを捨てるって方法も使うことがあるけど．

B君　：どういうことですか．

N先輩：たとえば，12ビットのA-Dだったら，下位の1ビットを捨てて上位11ビットの値を使うとか，下位の2ビットを捨てて上位10ビットの値を使うとかいう方法だよ．

B君　：それってなんかもったいない気がしますね．

〔図5〕ダブルバッファ




N先輩：分解能を下げることになるからもったいないといえばもったいないけど，下位ビットはノイズだとみなせば捨てるってことにもそれなりの意味があるんだよ．

B君　：そうなんですか．今回は6個のうちの最大と最小を捨てて4個の平均を取る方法を使ってみようと思います．

## 実行速度を上げるためには

N先輩：デバッグも終盤にさしかかってくると実行速度が気になり始めるものなんだ．実行速度を上げるためにはアルゴリズムの見直しやアセンブラへの書き換えなんかが必要になるけど，闇雲にやっても効果があるかどうかがわからないから，その前にプログラムを解析しないといけないんだ．そのためにはプロファイラと呼ばれるユーティリティが用意されていることがあるから，利用するといいよ．

B君　：はい．

N先輩：実際の使い方の説明はまた今度してあげるよ．

**● ダブルバッファ――ポインタを差し替える**

B君　：もう一つついでに高速化の方法で相談にのってください．

N先輩：ああ，いいよ．

B君　：Z80のアプリなんですけど，割り込みで100点ほどサンプリングして，サンプリングが終ったら，通常ルーチン用のバッファにコピーして，ある種の演算をさせてるんです．1点が16ビット×2なんで4バイトだから，400バイトほどLDIR命令で転送すると，4MHzのZ80の場合，

400×21クロック/4MHz→2100μsec

かかるんですよ．これが一つならいいんですけど，同じような割り込みが三つもあるもので，時々おかしくなるんですよ．ここを何とか高速化したいんですけど，うまい方法はないですかね．

N先輩：ああ，そういうときは通常，ダブルバッファにするんだよ．

B君　：どういうことですか．

N先輩：今は割り込みルーチン用と通常ルーチン用とで二重のバッファをもってるだろ．

B君　：はい．

N先輩：それを図5のようにポインタで差し替えるんだよ．割り込みルーチン用が差しているバッファがいっぱいになったら，サンプリング完了フラグをセットして，ポインタを通常ルーチン用と交換させるんだよ．そうすればポインタの差し替えだけだから，バッファ全体を転送させる必要がなくなって高速にできるはずだよ．

B君　：あっ，そうですね．通常ルーチンの処理は次のサンプリングが終わるまでに済んでますから，これだとうま

くいきそうですね．

● シフトアップ→ポインタのチェーン

N先輩：ついでに言っておくと，うちの製品だと500とか1000とかのオーダを管理して，一つのオーダの生産が終わるとシフトアップという処理をするんだけど，これも単純にシフトアップさせると数百オーダのバッファの転送になるから，32ビットCPUを使っても相当の時間がかかることになるんだ．

B君　：1オーダ100バイトとしても数十Kバイトですからね．

N先輩：実際は1オーダ1Kバイト程度だから，数百Kバイトなんだよ．こういうときは1オーダのマップの中に次オーダへのポインタをもたせておいて，シフトアップの時に図6のようにポインタのチェーンを書き換えるだけにすれば高速にできるんだ．

B君　：ええ，そうですね．

N先輩：シフトアップしたオーダを空きの先頭にもっていくか，最後にもっていくかという検討項目は残ってるけどね．Undoのことを考えたら，空きオーダの最後にもっていくほうが良いけど，ポインタのチェーンを最後までたどるとそこそこの時間がかかるからね．場合によっては次オーダだけでなく，前オーダへのポインタももたせて双方向にたどれるようにしておく必要があるかもしれないね．

B君　：双方向にする場合は最終オーダへのポインタももっていればいいんですね．

N先輩：1000オーダもあるとポインタをたどって検索していくだけでも，1オーダ1μsとしても最大1msもかかることになるよね．

B君　：そうですね．

N先輩：制御だとmsオーダは遅いほうだから，最後から探したほうが速いとわかる場合は，最後から探せるようにポインタを双方向にしておいたほうが良い場合もあるんだ．

B君　：ええ，そうでしょうね．

N先輩：高速化っていうとアセンブラ化するってことが，すぐに思い浮かぶけど，Cをアセンブラ化しても2倍になれば良いほうで2～3割しか速くならないことも多いから，アセンブラ化の前に考え方（アルゴリズム）を検討したほうがいいんだよ．

B君　：はい，わかりました．

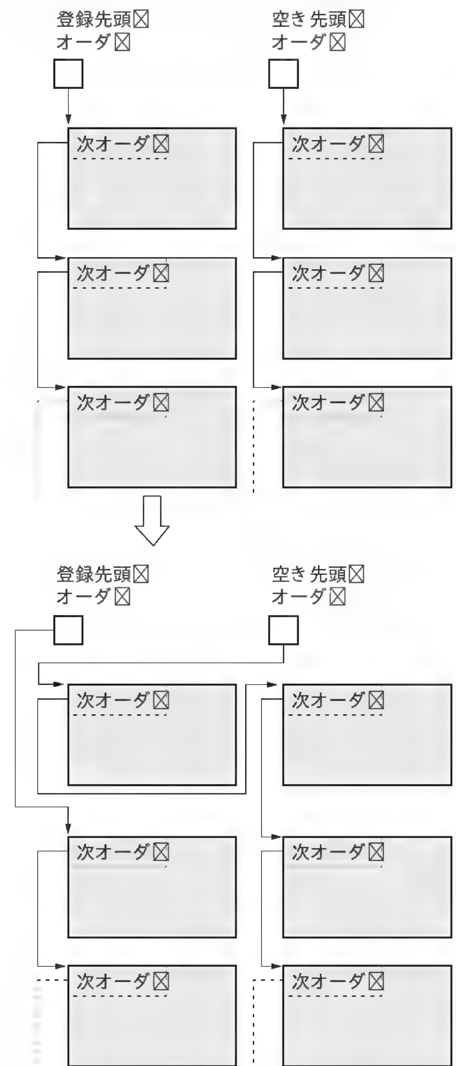
● インラインアセンブラ
——C/C++で記述できない部分を記述

N先輩：アセンブラが出たついでにインラインアセンブラについてちょっと言っておこうか．

B君　：インラインアセンブラと普通のアセンブラ（MASM，TASM）とどちらが良いんですかね．

N先輩：どちらにも一長一短があって，一概には言えないよ．

〔図6〕シフトアップ→ポインタのチェーン




最近はC/C++が主流でアセンブラは補助的にしか使われないから，自分にあったほうを使えば良いと思うよ．もっとも，MASMやTASMはそれなりの下地がないと使えないから，B君なんかはC/C++のインラインアセンブラから使ってみれば良いんじゃないかな．

B君　：ちょっと，使い方を教えてもらえませんか．

N先輩：まず，原則として言っておくと，インラインアセンブラを使うとC/C++の最適化に悪影響を及ぼすから，むやみに使わないほうが良いよ．基本的にインラインアセンブリコード自体は最適化の対象外になるし，インラインアセンブリコードがある関数全体でレジスタ割り付けやループの最適化なんかが抑制されるから，インラインアセンブラを使う部分はなるべく別関数にしておくほうが良いと思うよ．最近のコンパイラの最適化性能は目を見張るものがあるから，へたにインラ

インアセンブラを使うと生成コードが悪くなるってことにもなりかねないんだ．

B君　：それって，逆効果ですね．

N先輩：だから，インラインアセンブラの用途の一番目は速度アップよりも，C/C++で記述できない部分をインラインアセンブラで記述するってことだと思うよ．たとえば，簡単なフィードバック制御をするときなんかに最終的な出力値が急激に変化するのを避けるために，

$$I_{dat} \times \frac{I_{c1}}{I_{c2}}$$

といった演算を行って，61/100とか7/15とか適当に抑えることがあるんだけど，ldatがunsigned longだとldat * lc1が32ビットを超えてオーバフローする可能性があるよね．

B君　：そうですね．

N先輩：32ビットの86系のアセンブラの掛算や割算は，

32ビット × 32ビット → 64ビット

64ビット ÷ 32ビット → 32ビット

だから，上記のような掛け算と割り算だったら，オーバフローさせずに計算することができるんだ．

B君　：でも，そんなときは浮動小数点演算をすれば良いんじゃないですか．

N先輩：確かに，最近のPentiumプロセッサのようにコプロセッサが内蔵されてて浮動小数点演算が高速な場合は

〔リスト13〕32ビット←32ビット*32ビット/32ビット

```
#if   defined(_MSC_VER)      /* MS-C/C++, VC++   */
#if   _MSC_VER <= 600        /* MS-C 6.00A       */
#define __asm   _asm
#endif
#elif defined(__BORLANDC__) /* BC++             */
#define __asm   _asm
#elif defined(__TURBOC__)   /* Turbo C/C++      */
#include <dos.h>
#define __asm   asm
#endif

unsigned long lmuldiv(unsigned long ldat,      /* データ */
                      unsigned long lc1,       /* 分子 */
                      unsigned long lc2)       /* 分母 */
{                             /* ldat * lc1 / lc2     */
    unsigned long lans;

#if   _MSC_VER >= 900        /* VC++ 2.0 以降   (32ビット) */
    __asm mov eax,ldat
    __asm mul lc1
    __asm div lc2
    __asm mov lans,eax
#elif defined(__TURBOC__) /* TC, TC++, BC++  (16ビット) */
    __emit__((char)0x66); __asm mov ax,word ptr ldat
    __emit__((char)0x66); __asm mul word ptr lc1
    __emit__((char)0x66); __asm div word ptr lc2
    __emit__((char)0x66); __asm mov word ptr lans,ax
#else                        /* MS-C/C++, VC++  (16ビット) */
    __asm _emit 0x66 __asm mov ax,word ptr ldat
    __asm _emit 0x66 __asm mul word ptr lc1
    __asm _emit 0x66 __asm div word ptr lc2
    __asm _emit 0x66 __asm mov word ptr lans,ax
#endif
    return lans;
}
```

B君の言うとおりだけど．昔は80386＋80387の浮動小数点演算が遅かったから，固定小数点方式で整数演算をしていたんだよ．だから，インラインアセンブラの一つの用途として聞いてよ．

B君　：はい，わかりました．

N先輩：上記の演算を関数にするとリスト13のようにかけるんだ．リスト13はいろいろなコンパイラをサポートしたから，見にくくなってるけどね．

B君　：そうですね．

N先輩：VC++ 2.0以降って部分は32ビットコンパイラの例だけどこの部分がいちばん素直に書けてるよ．

```
__asm mov eax,ldat
__asm mul lc1 /* ldat * lc1
                      -> edx:eax   */
__asm div lc2 /* ldat * lc1 / lc2
                         -> eax */
__asm mov lans,eax
```

引き数のldat，lc1，lc2やローカル変数のlansがそのまま使えるから，なんとなくわかると思うけど．

B君　：はい，なんとなくわかります．

N先輩：__asmキーワードはMS-C 6.0AやBC++ 3.0だと_asmでTC，TC++だとasmになるんだけど，本質じゃないから気にせずにリストを眺めて見てよ．

B君　：はい．

N先輩：16ビットコンパイラだと386以降の32ビット命令がサポートされてないから，ちょっとした小細工が必要になるんだ．

B君　：確かに，16ビットの部分は良くわかんないですね．

N先輩：32ビット命令だとプレフィックス66hを付加するとオペランドのサイズが切り替わって，16ビットモードで動作してるときは32ビットサイズになるんだ．だから，__emit__((char)0x66)や__asm _emit 0x66はMASM，TASMだとDB 66hに相当する疑似命令で右側に記述したコードを32ビットサイズに変更することになるんだ．

B君　：word ptrというのはワードだから16ビットサイズになるはずなんだけど，プレフィックス66hを付加してるから，実際は32ビットサイズになるってことですか．

N先輩：そうだよ．MASM，TASMを使えば32ビット命令がそのまま使えるから，こんなのはMASM，TASMで書いたほうがわかりやすくなるんだけど，C/C++でかけない例の一つとして頭に置いておくといいよ．

B君　：はい．

N先輩：プレフィックス66hを指定するところが0x66となってるけど，どちらでもかまわないんだ．だから，定数を#defineで定義したマクロをインラインアセンブラの部分でも使うこともできるんだ．

B君　：それって便利ですね．

N先輩：くれぐれも，調子にのって，インラインアセンブラを使いすぎないようにね．僕はその昔，グラフィックのビット演算を500行くらいインラインアセンブラで書き直したことがあるんだけど，ほかの処理のほうが支配的でほとんど速度が変わらなかったことがあるんだ．遊びのユーティリティでやったことだから，おもしろ半分だったんだけどね．

B君　：はい，わかりました．

## 24ビットカウンタを32ビットに拡張する(正転，逆転あり)

B君　：もう一つ，相談にのってほしいんですが．

N先輩：なんだい．

B君　：このボードのファームウェアは先輩が作ったんですよね．でもカウンタICが製造中止になって新しいカウンタICに差し替えないといけないんですよ．

N先輩：その話は僕もちょこっと聞いたけど，僕が別件で忙しいからB君のところにいったんだね．

B君　：そうなんですよ．元々のカウンタは32ビットなんですが，新しいカウンタは32ビットのものが見つからなくて24ビットになったんですよ．でも，アプリケーションとのインターフェースは変えられないから，ハードウェアの24ビットカウンタを，ソフトウェアで32ビットに拡張しないといけないんですよ．それで，デバッグするとおかしいんですよ．ちょっと，デバッグにつき合ってもらえませんか．

N先輩：ああ，いいよ．

《試験室にて》

B君　：これを見てください．カウンタ値をコンソールに出力させているんですが，なんかおかしいと思いませんか．

N先輩：なんか，ちょっとおかしいね．プログラムを見せてごらん．

B君　：これですが．

N先輩：うーむ…B君，これって正転のときしか考えてないんじゃない．

B君　：ええ，そうですよ．

N先輩：このカウンタは2相入力が入ってるから，逆転も考慮しないといけないよ．

B君　：あっ，そうなんですか．この間，8ビットカウンタをやったときはインクリメントしか考えなかったから，まったく頭に入ってませんでした．

N先輩：単相入力のカウンタだったら，それで問題なかったんだけど，このカウンタは単相，2相のどちらも使えるんだ．たまたま，この試験装置が逆相の入力になってたから，下位24ビットがダウンカウントして，最上位バイトがカウントアップしているから，異常な速度で最上位バイトが桁上がりしてカウンタの表示がおかしかったんだよ．

〔リスト14〕24ビットカウンタ→32ビットカウンタ拡張

```
#define MAXCNTDIF   0x100000L  /* 最大カウント差 (正転／逆転判定用)*/

    lastcnt = 前カウンタ値(32ビット);
    last24  = lastcnt & 0x00ffffffL;  /* 前カウンタ値 (24ビット) */
    lnow24  = 現カウンタ値(24ビット);
    ihigh   = 0;                      /* 最上位バイト補正       */
    if (lnow24 > last24) {
        if (lnow24 - last24 >= MAXCNTDIF)
            ihigh = -1;               /* 000000→FFFFFF：逆転 */
    } else if (lnow24 < last24) {
        if (last24 - lnow24 >= MAXCNTDIF)
            ihigh = 1;                /* FFFFFF→000000：正転 */
    }
    現カウンタ値(32ビット) = lnow24 |
        ((((lastcnt >> 24) + (unsigned char)ihigh) & 0xffL)
<< 24);
```

B君　：試験装置が運良く(？)間違っていて良かったですよね．でも，どうすればいいんですか．

N先輩：桁上がりだけでなくて，桁下がりも考慮したプログラムに変更すればいいんだよ．具体的には前カウンタ値の下位24ビットと現カウンタ値の下位24ビットを比較して，現カウンタ値のほうが増えていれば，その差がある値(たとえば，100000h：絶対に起こり得ない差)以上だったら，逆転といった判断になるかな．たとえば，

　前カウンタ値：000100h
　現カウンタ値：FF0000h

だったら，その差はFEFF00hだから逆転したと判断するんだよ．

B君　：あっ，そうですか．だったら，正転の判断は逆に現カウンタ値のほうが減っていれば，

　前カウンタ値：FF0000h
　現カウンタ値：000100h

で，その差がFEFF00hだから正転したという判断になるんですね．

N先輩：そうだね．だから，まとめるとリスト14のようになるんじゃないかな．

B君　：うーむ…そうですね．それじゃ，このように書き直してやってみます．

## 場所によって使用して良い関数，いけない関数

N先輩：ライブラリに用意されている関数は当然，すべて使用可能な関数なんだけど，

- 割り込み処理

●常駐プログラムの常駐部
●デバイスドライバ

なんかでは使用できる関数は限られているんだ．これらの部分だとOSの機能を使用している関数は使用できないけど，ハードウェア割り込みがからむ部分だと再入可能でない関数も使用できないんだ．上記の部分で使用可能な一般的な関数をあげてみると，

(1) ctype.h，jctype.hのマクロ
(2) memxxx関数(memchr，memcmp，memcpy，memset，etc)
(3) strxxx関数(strcat，strchr，strcmp，strcpy，strlen，etc)
(4) 算術変換関数(abs，atoi，atol，itoa，labs，etc)
(5) その他(bsearch，qsort，tolower，toupper，inp，outp，etc)

という感じかな．

N先輩：いろいろと思いつくままに話したから，つながりがなくてわかりにくかったと思うけど．プログラムを書いた本人がデバッグすると無意識のうちに動くように操作してしまうことも少なくないんだ．デバッグを始める前にはデバッグする項目を拾い上げるとともに，プログラムを書いた自分と別の方向から見ることができるように頭を切り替えることも必要かもしれないね．今日はこれくらいにしておこうか．

B君　：はい，ありがとうございました．

## おわりに

本章ではデバッグに関して概論を述べただけで具体的に説明できませんでした．割り込み処理などのデバッグが難しい実際のデバッグ手法に関しては別の機会に説明したいと思います．

なかしま・のぶゆき　(株)Unix

第5章

ROM化するということはどういうことか

# 組み込みCプログラミング

中島 信行

Cではポインタと配列は，コーディングする上では同じように記述できる場合があります．Cらしさではポインタによるコーディングということになりますが，読みやすさの点では配列でのコーディングが良いこともあります．本章ではコーディングの違いによって，最適化がどのように影響を受けるのか，いくつか例をあげてみていきたいと思います．（筆者）



## printf("hello, world¥n");を動かすには

C君　：先輩，おはようございます．

N先輩：おはよう．

C君　：今度，例の組み込み機器の開発をする一員になったんですけど，いろいろとアドバイスをお願いします．

N先輩：C君はWindows上のアプリケーションしか作ったことがないんだっけ．

C君　：そうなんです．

N先輩：組み込み機器はパソコンと違って，いろいろと制約があるからね．たとえば，リスト1のプログラムはわかるかい．

C君　：C言語の本の最初に載ってるやつですよね．でも，Windows上のアプリケーションしか作ったことがないんで，試したことはないですけど．

N先輩：えーっ，そうなんだ．今，Linux端末を使ってるから，リスト1を打ち込んで，コンパイルして，

```
$ gcc hello.c -o hello
```

を実行すると，

```
$ ./hello
hello, world
$
```

ほら，hello, worldと表示したよね．

C君　：Linuxはやったことがないんで，細かいことはわからないですけど，GCCというコンパイラでコンパイルして，実行してみたってことですよね．

N先輩：そうなんだ．だったら，Windowsのコマンドプロンプトでリスト1を打ち込んで，コンパイルして，

```
C:¥>cl hello.c
```

実行すると，

```
C:¥>hello
hello, world
C:¥>
```

ほら，hello, worldと表示したよね．細かいことは気にしなくていいけど，Windowsのコンソールアプリケーションや MS-DOS（以下，DOSと略す），Linuxなんかの OSのある環境で Cプログラムを作成する場合は，main関数から実行が始まるから，リスト1のように単純にプログラムが書けるけど，組み込み機器の場合は，リスト1を動かすまでにはいろいろとやっておかないといけないことがあるんだ．

C君　：へぇー，そうなんですか．printfって言ったらCの標準関数ですよね．Cコンパイラを使ったら，どの環境でも使えるんじゃないんですか．

N先輩：使えると言っちゃ，使えるんだけど，使えるようにするためにいろいろとやらないといけないことがあるんだ．

## ROM化のレベルはターゲットによって違う

N先輩：パソコンの場合は，電源スイッチをいきなり切ってはいけないことになってるから，いきなり電源を切ってハードディスクのファイルが壊れてもユーザーの責任ということになるけど，組み込み機器は電源をいきなり切られるのが普通だよね．

C君　：そりゃそうですね．

〔リスト1〕C言語の本の最初に載っているプログラム（hello.c）

```
#include <stdio.h>

int main()
{
  printf("hello, world¥n");
  return 0;
}
```




N先輩：そのためには，ROM化が必須条件になるんだ．C君はROM化についてははじめてだろうから，簡単に説明していこう．

C君　：はい，お願いします．

N先輩：ひと口にROM化といっても，いろいろなレベルがあって，

● OSごとROM化する
● ROM化対応のOSを使ってROM化する
● OSと切り離してプログラムだけをROM化する

てな感じに分けられるかな．

C君　：もう一つピンとこないんですが．

### ● OSごとROM化する

N先輩：OSごとROM化するってのは，OSをROMディスクに書き込んでROMディスクから自動的に立ち上げる方法なんだ．

C君　：ROMディスクって何ですか．

N先輩：うーむ．C君はRAMディスクって知ってるかい．

C君　：知りませんけど．

N先輩：困ったな．Windows上のプログラムから始めると何も意識しなくてもプログラムが組めちゃうからね．それじゃ，パソコンがハードディスクやフロッピディスクから起動するってのは知ってるよね．

C君　：いくらなんでも知ってますよ．CD-ROMからも起動できますよね．

N先輩：おーっ，良いこと言ったね．昔のパソコンはね，CD-ROMから起動できなかったんだよ．

C君　：えーっ，そうなんですか．

N先輩：昔のパソコンは，BIOSがCD-ROMからの起動をサポートしていなかったから起動できないんだ．ROMディスクも同じことで，BIOSがサポートしていないとだめなんだけど．小さなプログラムだとフロッピに書き込んで，フロッピで起動して実行することができるだろ．

C君　：はい．

N先輩：そのフロッピの代わりにROMを使うのがROMディスクなんだ．ROMディスクはプログラムをファイル形式でROMに書き込んでおいて，アプリケーションプログラムを実行ファイルとしてROMディスクからRAM上にロードして実行するんだ．

C君　：ファイル形式でROMに書くってどういうことですか．

N先輩：ふーむ．C君はWindows 98を使ったことがあるから，FATってのは知ってるよね．

C君　：FAT32とかいうやつですよね．

N先輩：そうだよ．簡単に言うと，FATはディスクの最初のほうにディスク上の位置を指すFAT領域やルートディレクトリ領域なんかがあるんだけど，ファイルを作成するとルートディレクトリ領域にディレクトリが作成されて，FAT領域にそのファイルのディスク上の位置が書き込まれるんだ．ROMをこれに習って，書き込んでいけばディスクとして扱えるようになるんだ．

C君　：なんとなくわかりかけてきました．

N先輩：ROMディスクのファイル形式はFATだけとは限らないけどね．OSごとROM化する方式はパソコンでプログラムを実行するのと同じ感じになるから，プログラミングに関してはとくに注意することはないんだ．

C君　：だったら，OSごとROM化しちゃえば簡単ですね．

N先輩：そうなんだけど，OSごとROM化する方法はROMに書き込んであるプログラムをいったん，RAM上にロードして実行することになるから，メモリが余分に必要になるんだ．といっても，最近は，メモリが大容量になってきたから，うちのように量産品を作らない場合はとくに問題にならないことも多いんだけどね．

C君　：そうですよね．

N先輩：もう一つ，問題はどうやってOSごとROM化するかってことなんだ．

C君　：そんなに難しいんですか．

N先輩：OSごとROM化するようなシステムはたいていパソコンやボードパソコンを使うんだけど，そのBIOSがROMディスクを認識してくれないといけないから，自分で全部やろうとすると最初はたいへんだと思うよ．でも，MS-DOS，Windows，Linuxなんかだと OSごとROM化できるシステムが市販されてるから，たいていは市販品を使うことになると思うけどね．

C君　：だったら，難しくないんでしょうね．

N先輩：まあ，そうだね．

### ● ROM化対応のOSを使ってROM化する

N先輩：ROM化対応のOSを使ってROM化する方法はOS自体がROM上で実行できるようになっているんだ．だから，メモリの使用効率は良くなるんだけど，ROMはアクセス速度が遅いから，RAMにロードできるよ

うになっているものもあるけどね．

C君　：ROMって遅いんですか．

N先輩：パソコンがRAMをたくさん使うから，RAMはどんどん進化して，大容量で高速なアクセスができるものがこれからもでてくるだろうけど，ROMはたいていは起動時にしか使われないから，遅くても問題はないから，あまり進化しないんだ．

C君　：そういうもんでしょうね．

N先輩：アプリケーションはROMディスクの形式で書き込めばOSごとROM化する方式と同じでプログラミングに関してはとくに注意することはなくなるね．この方式は僕が実際にやったことがないから，これぐらいにしておこう．

C君　：はい．

### ● OSと切り離してプログラムだけをROM化する

N先輩：一般的にROM化っていうと，OSと切り離してプログラムだけをROM化する方式のことを指すんだけど．OSはプログラムを開発する環境として利用して，ターゲットのシステムは最終的な実行ファイルをSレコードやHEXフォーマットに変換して，ROMに書き込むんだ．

C君　：何を言われてるのか，わからないんですけど…．

N先輩：それじゃ，OS上の普通のプログラムとROM化プログラムの違いを簡単に説明しようか．リスト2を見てごらん．これの問題点がなんだかわかるかい．

C君　：いいえ，わかりません．

N先輩：ちょっと，質問が悪かったかな．たとえば，変数dataがROMに割り付けてあったら，どうだい．

C君　：ROMだったら，書き換えできませんよね．

N先輩：そうだね．初期値をもつ変数をROMに割り付ける場合は書き換えないようにしないといけないんだ．言葉で言うと簡単だけど，たとえば，8086用のコンパイラだと，普通に記述すると初期化されていない変数領域と初期値をもつ変数領域が64Kバイト内の範囲に割り付けられるから，

- 初期化されていない変数領域：RAM
- 初期値をもつ変数領域：ROM

ってな具合に分けるのが難しくなるから，初期値をもつ変数領域を離れた領域に割り付けるようにしないといけない場合があるんだ．

C君　：そうなんですか．

N先輩：それじゃ，書き換えるには，どうすれば良い?

C君　：RAMに割り付ければ良いんじゃないですか．

N先輩：そうだね．でも，RAMにしたら，起動時の内容は不定だよね．

C君　：あっ，そうですね．だったら，どうするんですか．

N先輩：めったに書き換えない定数だったら，書き換えのできるEEPROMやフラッシュメモリに割り付けるっていう方法があるけど，一般的じゃないよね．ROM化はコンパイラがある程度対応してくれてないといけないんだけど，リスト3のように初期化関数を作って，初期値が必要な変数に初期値を設定する方法だったら，コンパイラが生成するメモリ配置を調べる必要がないから，まず，問題なく適用できると思うよ．でも，初期値をもつ変数が多いと面倒だから，一般的には，初期値をROMに書いておいて，起動時に，その初期値を「初期値をもつ変数の領域」に転送するようにするんだ．

〔リスト2〕初期値を書き換えるプログラム例1

```
int data = 1;

void func(void)
{
  data = 2;
}
```

〔リスト3〕初期化関数

```
int data1;
int data2;
/* ... */
int datan;

void init(void)
{
  data1 = 1;
  data2 = 2;
  /* ... */
  datan = 10;
}
```

〔リスト4〕自動変数を書き換えるプログラム例

```
void func(void)
{
  int data = 1;

  data = 2;
}
```

〔リスト5〕初期値を書き換えるプログラム例2

```
void func(void)
{
  static int data = 1;

  data = 2;
}
```

C君　：なるほど．

N先輩：でも，この方法はコンパイラが生成するメモリ配置を調べないといけないから，コンパイラによっては難しいこともあるんだ．

C君　：そうなんですか．

N先輩：初期値をもった変数領域のアドレスとバイト数がわからないとどうしようもないだろ．

C君　：そうですね．

N先輩：いずれにしても，ROMは書き換えできないから，ROM化システムの基本はROM領域とRAM領域を分けることなんだ．それじゃ，簡単な問題を出そう．リスト4に何か問題はあるかな．

C君　：えっー，わかりません．

N先輩：リスト4の変数dataは自動変数だから，何も問題はないんだ．自動変数をRAMに割り付ければ良いからね．それじゃ，もう一つ，リスト5に何か問題はあるかな．

C君　：リスト5の変数dataは初期値をもつ変数だから，

〔図1〕リスト1のMAPファイル(VC++ 1.5)

```
Start  Stop   Length Name               Class
00000H 014D7H 014D8H _TEXT              CODE
014E0H 014E1H 00002H EMULATOR_TEXT      CODE
014E2H 014E2H 00000H C_ETEXT            ENDCODE
014F0H 014F0H 00000H EMULATOR_DATA      FAR_DATA
014F0H 01531H 00042H NULL               BEGDATA
01532H 0179BH 0026AH _DATA              DATA
0179CH 0179DH 00002H XIQC               DATA
0179EH 017A9H 0000CH DBDATA             DATA
017AAH 017B7H 0000EH CDATA              DATA
017B8H 017B8H 00000H XIFB               DATA
017B8H 017B8H 00000H XIF                DATA
017B8H 017B8H 00000H XIFE               DATA
017B8H 017B8H 00000H XIB                DATA
017B8H 017B8H 00000H XI                 DATA
017B8H 017B8H 00000H XIE                DATA
017B8H 017B8H 00000H XPB                DATA
017B8H 017B9H 00002H XP                 DATA
017BAH 017BAH 00000H XPE                DATA
017BAH 017BAH 00000H XCB                DATA
017BAH 017BAH 00000H XC                 DATA
017BAH 017BAH 00000H XCE                DATA
017BAH 017BAH 00000H XCFB               DATA
017BAH 017BAH 00000H XCFCRT             DATA
017BAH 017BAH 00000H XCF                DATA
017BAH 017BAH 00000H XCFE               DATA
017BAH 017BAH 00000H XIFCB              DATA
017BAH 017BAH 00000H XIFU               DATA
017BAH 017BAH 00000H XIFL               DATA
017BAH 017BAH 00000H XIFM               DATA
017BAH 017BAH 00000H XIFCE              DATA
017BAH 017BAH 00000H CONST              CONST
017BAH 017C1H 00008H HDR                MSG
017C2H 01897H 000D6H MSG                MSG
01898H 01899H 00002H PAD                MSG
0189AH 0189AH 00001H EPAD               MSG
0189CH 0189CH 00000H _BSS               BSS
0189CH 0189CH 00000H XOB                BSS
0189CH 0189CH 00000H XO                 BSS
0189CH 0189CH 00000H XOE                BSS
0189CH 0189CH 00000H XOFB               BSS
0189CH 0189CH 00000H XOF                BSS
0189CH 0189CH 00000H XOFE               BSS
018A0H 0209FH 00800H STACK              STACK

Origin   Group
014F:0   DGROUP

 Address         Publics by Name

014F:03AC       _edata
014F:03B0       _end
014F:008C       _errno
0000:0200       _exit
0000:0672       _fclose
0000:0972       _fflush
0000:0010       _main
0000:0728       _printf
0000:13E4       _remove
014F:00CA       _STKHQQ
0000:127A       _strcat
0000:12BA       _strcpy
014F:00C8       __aaltstkovr
014F:0070       __acfinfo
0000:9876  Abs  __acrtmsg
0000:9876  Abs  __acrtused
0000:D6D6  Abs  __aDBdoswp
014F:00C6       __adbgmsg
014F:0052       __aexit_rtn
014F:007E       __aintdiv
014F:0294       __amblksiz
0000:0103       __amsg_exit
0000:02F0       __aNchkstk
014F:0042       __anullsize
014F:006A       __aseghi
014F:006C       __aseglo
014F:0054       __asizds
014F:02A8       __asizeC
014F:02A9       __asizeD
0000:0032       __astart
014F:0050       __atopsp
0000:020F       __cexit
014F:0296       __cfltcvt_tab
014F:00E6       __cflush
014F:00BD       __child
0000:02F0       __chkstk
0000:012E       __cinit
0000:00F4       __cintDIV
0000:0F8A       __close
0000:1384       __cltoasub
0000:132C       __commit
014F:0099       __cpumode
0000:028E       __ctermsub
0000:1390       __cxtoa
0000:0219       __c_exit
0000:012C       __dataseg
014F:009A       __doserrno
0000:061A       __dosret0
0000:062F       __dosretax
0000:0622       __dosreturn
0000:13F2       __dos_commit
014F:03AC       __edata
014E:0000       __EmDataSeg
014F:03B0       __end
0000:0764       __endstdio
014F:00B6       __environ
0000:0207       __exit
014F:00C5       __exitflag
014F:0082       __fac
0000:0670       __farstub
0000:1308       __fcloseall
014F:006E       __fDosExt
0000:02CA       __FF_MSGBANNER
0000:1259       __findlast
0000:0776       __flsbuf
0000:09C0       __flush
0000:0A36       __flushall
014F:02BC       __fpinit
0000:02EA       __fptrap
0000:0856       __freebuf
0000:0933       __ftbuf
0000:0880       __getbuf
0000:117C       __growseg
0000:1208       __incseg
014F:00C0       __intno
014F:00E8       __iob
014F:0188       __iob2
0000:12EC       __itoa
014F:0228       __lastiob
0000:0FAA       __lseek
014F:0292       __lseekchk
0000:063C       __maperror
0000:05F6       __myalloc
014F:009C       __nfile
0000:1400       __nfree
014F:0056       __nheap_desc
0000:1421       __nmalloc
0000:0596       __NMSG_TEXT
0000:05C1       __NMSG_WRITE
0000:0308       __nullcheck
014F:009A       __oserr
014F:009E       __osfile
014F:0095       __osmajor
014F:0094       __osminor
014F:0098       __osmode
014F:0094       __osver
014F:0096       __osversion
0000:0AB4       __output
014F:00BF       __ovlflag
014F:00C1       __ovlvec
014F:00B8       __pgmptr
014F:02AA       __pnhNearHeap
014F:0092       __psp
014F:0090       __pspadr
0000:145C       __searchseg
0000:032A       __setargv
0000:0518       __setenvp
014F:02A6       __sigintoff
014F:02A4       __sigintseg
0000:116A       __stackavail
0000:08C2       __stbuf
014F:008E       __umaskval
0000:13E4       __unlink
0000:102C       __write
014F:02B4       ___aDBexit
014F:02B2       ___aDBrterr
014F:02B0       ___aDBswpchk
014F:02AE       ___aDBswpflg
014F:00B2       ___argc
014F:00B4       ___argv
014F:02AC       ___qczrinit

 Address        Publics by Value

0000:0010       _main
0000:0032       __astart
0000:00F4       __cintDIV
0000:0103       __amsg_exit
0000:012C       __dataseg
0000:012E       __cinit
0000:0200       _exit
0000:0207       __exit
0000:020F       __cexit
0000:0219       __c_exit
0000:028E       __ctermsub
0000:02CA       __FF_MSGBANNER
0000:02EA       __fptrap
0000:02F0       __aNchkstk
0000:02F0       __chkstk
0000:0308       __nullcheck
0000:032A       __setargv
0000:0518       __setenvp
0000:0596       __NMSG_TEXT
0000:05C1       __NMSG_WRITE
0000:05F6       __myalloc
0000:061A       __dosret0
0000:0622       __dosreturn
0000:062F       __dosretax
0000:063C       __maperror
0000:0670       __farstub
0000:0672       _fclose
0000:0728       _printf
0000:0764       __endstdio
0000:0776       __flsbuf
0000:0856       __freebuf
0000:0880       __getbuf
0000:08C2       __stbuf
0000:0933       __ftbuf
0000:0972       _fflush
0000:09C0       __flush
0000:0A36       __flushall
0000:0AB4       __output
0000:0F8A       __close
0000:0FAA       __lseek
0000:102C       __write
0000:116A       __stackavail
0000:117C       __growseg
0000:1208       __incseg
0000:1259       __findlast
0000:127A       _strcat
0000:12BA       _strcpy
0000:12EC       __itoa
0000:1308       __fcloseall
0000:132C       __commit
```

〔図1〕リスト1のMAPファイル(VC++ 1.5)(つづき)

```
0000:1384        __cltoasub
0000:1390        __cxtoa
0000:13E4        __unlink
0000:13E4        _remove
0000:13F2        __dos_commit
0000:1400        __nfree
0000:1421        __nmalloc
0000:145C        __searchseg
014E:0000        __EmDataSeg
014F:0042        __anullsize
014F:0050        __atopsp
014F:0052        __aexit_rtn
014F:0054        __asizds
014F:0056        __nheap_desc
014F:006A        __aseghi
014F:006C        __aseglo
014F:006E        __fDosExt
014F:0070        __acfinfo
014F:007E        __aintdiv
014F:0082        __fac
014F:008C        _errno
014F:008E        __umaskval
014F:0090        __pspadr
014F:0092        __psp
014F:0094        __osminor
014F:0094        __osver
014F:0095        __osmajor
014F:0096        __osversion
014F:0098        __osmode
014F:0099        __cpumode
014F:009A        __oserr
014F:009A        __doserrno
014F:009C        __nfile
014F:009E        __osfile
014F:00B2        ___argc
014F:00B4        ___argv
014F:00B6        __environ
014F:00B8        __pgmptr
014F:00BD        __child
014F:00BF        __ovlflag
014F:00C0        __intno
014F:00C1        __ovlvec
014F:00C5        __exitflag
014F:00C6        __adbgmsg
014F:00C8        __aaltstkovr
014F:00CA        _STKHQQ
014F:00E6        __cflush
014F:00E8        __iob
014F:0188        __iob2
014F:0228        __lastiob
014F:0292        __lseekchk
014F:0294        __amblksiz
014F:0296        __cfltcvt_tab
014F:02A4        __sigintseg
014F:02A6        __sigintoff
014F:02A8        __asizeC
014F:02A9        __asizeD
014F:02AA        __pnhNearHeap
014F:02AC        ___qczrinit
014F:02AE        ___aDBswpflg
014F:02B0        ___aDBswpchk
014F:02B2        ___aDBrterr
014F:02B4        ___aDBexit
014F:02BC        __fpinit
014F:03AC        __edata
014F:03AC        _edata
014F:03B0        __end
014F:03B0        _end
0000:9876  Abs   __acrtmsg
0000:9876  Abs   __acrtused
0000:D6D6  Abs   __aDBdoswp

Program entry point at 0000:0032
```

リスト2と同じことになりますよね.

N先輩:そうだね.

## ROM化するための前準備

### ● MAPファイルの構成と作り方

N先輩:それじゃ,コンパイラの生成するメモリ配置を見るためにリスト1のMAPファイルをLinuxで作ってみようか….

```
$ gcc hello.c -o hello -Wl,-M
.....
```

MAPファイル(省略)がちょっとわかりにくいね.じゃあ,DOS版のコンパイラでMAPファイルを作ってみよう.

```
C:¥>cl hello.c /Zd /link /m/li
```

図1を見てごらん.リスト1にはmainとprintfしかないけど,MAPファイルには多くの識別子が出てきてるよね.VC++だと,変数名や関数名は先頭にアンダーライン(_)がつけられるから,mainは_mainになってるけどね.

C君 :そうですね.

N先輩:簡単にMAPファイルを説明すると初めにでてくるClassにCODE,DATAとかがあるよね.

C君 :はい.

N先輩:CODEはプログラムのコード部分のことで,FAR_DATAやDATAは初期値をもった静的変数,CONSTは定数,BSSは初期化値をもたない静的変数,STACKはスタック領域なんだ.DOS上のコンパイラは細かく分けてるけど,一般的にはプログラムコード部と変数部に分けられて,変数部は,

- 初期化値をもたない静的変数
- 初期値をもつ静的変数
- スタック領域

に分けられるんだ.ちなみに,Windows用のBorland C++ 5.6だと図2のようにClassがシンプルになるんだけど.

〔図2〕リスト1のMAPファイル(Borland C++ 5.6 for Win32)

```
     Start        Length     Name                    Class
 0001:00401000 0000092F0H _TEXT                      CODE
 0002:0040B000 0000025F0H _DATA                      DATA
 0003:0040D5F0 000000870H _BSS                       BSS
 0004:00000000 00000009CH _TLS                       TLS

 Address Publics by Name

  ~~~ 省略 ~~~
```

C君 :そうですね.

N先輩:リスト1のプログラムで,図1のようにたくさん識別子が出てくるのは,スタートアップルーチンがリンクされているからなんだ.

C君 :スタートアップルーチンってなんですか.

### ● スタートアップルーチンの中身

N先輩:図1の最後に,

```
Program entry point at 0000:0032
```

があるだろ.

C君 :はい.

N先輩:これはプログラム開始アドレスのことなんだけど,図1の後半のPublics by Valueのところを見ると,

```
0000:0010        _main
```

```
0000:0032          __astart
```

となっているだろ.

C君　：あっ，プログラム開始アドレスが _main じゃなくて，__astart になってますね.

N先輩：その部分がスタートアップルーチンで，main 関数はスタートアップルーチンから呼ばれているんだ（図3）．特定OS用のコンパイラの場合は，動作環境がわかっているから，あらかじめ用意されているスタートアップルーチンをリンクするだけで良いんだけど，組み込み機器の場合は，さまざまな動作環境があるから，ユーザーがスタートアップルーチンを作成してやらないといけないんだ.

C君　：スタートアップルーチンは何をやってるんですか.

## ● スタートアップルーチンの処理

N先輩：それを説明するために，リスト6を見てごらん．リスト6は意味のあるプログラムじゃないけど，var1 はどうなるかわかるかい.

C君　：確か，初期化されていない静的変数は初期値が0になるから，var1 は1になりますね.

N先輩：パソコンなどのOSのある環境で動かすとスタートアップルーチンがいろいろとやってくれてるから何も意識しなくても，そうなるんだけど，ROM化環境だと，

- 初期化されていない静的変数
- 初期値をもつ静的変数

はその状態になるようにスタートアップルーチンを作らなくちゃいけないんだ．だから，ROM化環境でも初期化されていない静的変数の領域を0にクリアする初期値をもつ静的変数の領域に初期値を転送するといった処理を行って，main 関数を呼び出さないといけないんだ.

C君　：少しだけ，わかってきました.

N先輩：OSのある環境だとプログラムのコード部分なんかも含めて全部がRAMだけど，ROM化環境だとROMは書き換えができないから，それなりに意識してプログラミングしないといけないんだ．Cのプログラムだったら，ROM領域には，

- プログラムのコード部分
- 定数
- 初期値をもつ静的変数の初期値

などを割り付けて，RAM領域には，

- 初期化されていない静的変数の領域
- 初期値をもつ静的変数の領域
- 自動変数が割り付けられるスタック領域

なんかを割り付けるんだよ.

C君　：初期値をもつ静的変数はROMとRAMにでてきましたけど….

N先輩：初期値をもつ静的変数は初期値自体はROMに置くけど，実際の変数はRAMに割り付けるということなんだ．スタートアップルーチンでその初期値を実際の変数領域にコピーすれば，静的変数が初期値をもつことになるんだ.

C君　：あっ，そうですね.

N先輩：スタートアップルーチンは縁の下の力持ち的なもので，

- スタックポインタの設定
- 割り込みベクタの設定
- 初期化されていない静的変数の領域を0にクリアする
- 初期値をもつ静的変数の領域に初期値を転送する
- ハードウェアの初期化
- ハードウェアのチェック
- 割り込み許可

なんかを行うんだ.

## ● スタックポインタの設定

N先輩：C君はCPUやアセンブラの知識はあるのかい.

C君　：いいえ，ありません.

N先輩：CPUによって，細かい点は異なるから，一般論になるけど，Cの関数を呼んだ後，呼び出し元に戻るのはなぜだかわかるかい.

C君　：そういうもんだと思ってましたが.

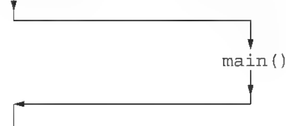
N先輩：まあ，そういうものなんだけど，呼び出し元に戻るためには，どこかに呼び出し元を記憶しておかないといけないんだ.

C君　：そうでしょうね.

N先輩：そのために，一般的にはスタックが使われるんだ．スタックポインタはスタックがどこまで使われているかを指し示すCPU内蔵のレジスタなんだけど．Cには記憶クラス指定子 register があるから，レジスタはわかるよね.

C君　：なんとなくわかってます.

〔図3〕スタートアップルーチンの流れ




〔リスト6〕初期値をもつ静的変数と初期化されていない静的変数

```
int data = 1;    /* 初期値を持つ静的変数 */
int var0, var1; /* 初期化されていない静的変数 */

int main()
{
  var1 = var0 + data;
  return 0;
}
```

〔リスト7〕絶対値を求める

```
int iabs(int data)
{
  int result;

  result = data;
  if (result < 0)
    result = -result;
  return result;
}
```

```
int main()
{
  int data1, data = -123;

  data1 = iabs(data);
  return 0;
}
```

〔図4〕リスト8のスタックフレーム

| | |
|---|---|
| ebp-4 | result |
| ebp+0 | ebp |
| ebp+4 | 戻り番地 |
| ebp+8 | data |

〔リスト8〕リスト7のアセンブリリスト(GCC：最適化なし)

```
    .file   "iabs.c"
    .version    "01.01"
gcc2_compiled.:
.text
    .align 4
.globl iabs
    .type    iabs,@function
iabs:
    pushl %ebp
    movl %esp,%ebp       フレームポインタのセット
    subl $4,%esp         int result;
    movl 8(%ebp),%eax
    movl %eax,-4(%ebp)   result = data;
    cmpl $0,-4(%ebp)     if (result < 0)
    jge .L2
    negl -4(%ebp)        result = -result;
.L2:
    movl -4(%ebp),%edx
    movl %edx,%eax
    jmp .L1
    .p2align 4,,7
.L1:
    leave
    ret
.Lfe1:
    .size    iabs,.Lfe1-iabs
    .align 4
.globl main
    .type    main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    movl $-123,-8(%ebp)
    movl -8(%ebp),%eax
    pushl %eax
    call iabs
    addl $4,%esp
    movl %eax,%eax
    movl %eax,-4(%ebp)
    xorl %eax,%eax
    jmp .L3
    .p2align 4,,7
.L3:
    leave
    ret
.Lfe2:
    .size    main,.Lfe2-main
    .ident  "GCC: (GNU) egcs-2.91.66 19990314/
                              Linux (egcs-1.1.2 release)"
```

N先輩：リスト7は絶対値を求める，ちょっと，わざとらしいプログラムだけど，これを，

`data1 = iabs(data);`

のように呼び出すと，引き数dataをスタックに積んだ後，戻り番地もスタックに積んで，iabsに実行を移すんだ．iabsにくると自動変数resultの領域がスタックに取られるんだ．RISC CPUの中には戻り番地用のレジスタが用意されているものもあるけど，関数を多重に呼び出すことを考えると，いずれにしても，戻り番地をスタックに積まないといけなくなるんだ．リスト8を見るとpushl %ebpでebpレジスタをスタックに退避しているけど，これはフレームポインタといって，自動変数をアクセスしやすくするためのものなんだ．図4を見ればわかるように引き数dataはebp+8～ebp+11になって，自動変数resultはebp-4～ebp-1になるんだ．こんなふうにスタックは，

- 関数の引き数
- 関数の戻り番地
- 自動変数

として使われるんだ．割り込みのときには戻り番地やレジスタなどのコンテキストの保存に使われるよ．RISC CPUだと戻り番地やレジスタが割り込み専用のレジスタに退避されるものもあるけど，多重割り込みを許可する場合は，これらをスタックに退避しておく必要があるんだ．だから，スタートアップルーチンではスタック領域の先頭アドレスをスタックポインタレジスタに設定しておかなければいけないんだよ．

### ●割り込みベクタの設定

C君　：割り込みにはどんなものがあるんですか．

N先輩：それこそ，組み込み機器ごとに違うけど，割り込みベクタはたとえば，0で割ったりしたら例外が発生して，メッセージを表示するだろ．

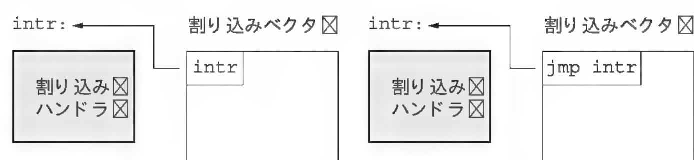
C君　：はい，何度か，やったことあります．

N先輩：後は，Ctrl+Cでプログラムをブレークするとか，OSのある環境だと，そのプログラム(通常はプロセスという)用の割り込みベクタをセットしてくれるんだけど．これらは内部割り込みと呼ばれているんだ．

C君　：内部というからには外部割り込みってのもあるんですか．

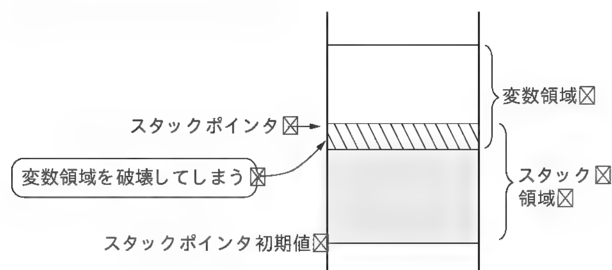
N先輩：ハードウェアからの割り込みを外部割り込みというんだ．組み込み機器の場合は内部割り込み以外に，タイマ割り込みは，まず，必要だから，そういったハードウェアに関連した外部割り込みのベクタを設定しておかなくちゃいけないんだ．割り込みベクタというと一般的には割り込みハンドラの先頭アドレスを書いておくんだけど，CPUによっては，割り込みベクタのアドレスにジャンプしてくるものもあるから，そこに，割り込みハンドラへのジャンプ命令を書かなくちゃいけない場合もあるんだ〔図5(a)(b)〕．タイマ割り込み

〔図5〕割り込みベクタ

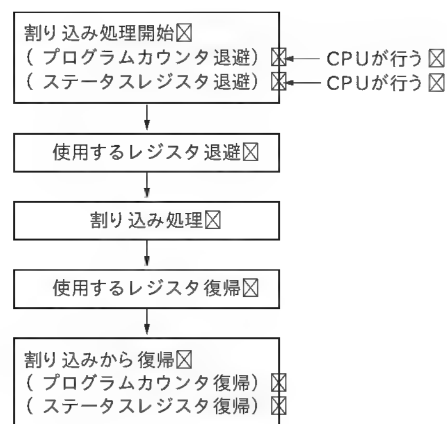



(a) 割り込みハンドラの先頭アドレス　(b) 割り込みベクタのアドレスにジャンプ

〔図7〕スタック領域が足らないと…




〔図6〕割り込みハンドラ




以外にも通信(RS-232-Cなど)の割り込みも一般的に必要になるよね.

C君　：通信はよく使いますからね.

N先輩：最初に言った，電源をいきなり切られたときに，そのときの設定状態を覚えておかないといけない機器の場合は，フラッシュメモリなどに書き込んでおかないといけないから，電源電圧の低下を検知する割り込みなども必要になるんだ.

C君　：そうですね.

N先輩：割り込みハンドラは図6のような流れになって，割り込みが発生すると元のプログラムに戻れるようにするため，CPUが，

- プログラムカウンタ
- ステータスレジスタ

をスタックまたは専用のレジスタに退避して，割り込み処理にジャンプしてくれるんだ．プログラムカウンタはそのとき実行している命令コードのアドレスを示していて，ステータスレジスタは演算や条件分岐のときなどに使用されるフラグなんかが収められているレジスタのことなんだ．割り込みハンドラでは最初に割り込み処理で使用するレジスタを退避してから，本来の割り込み処理を実行して，済んだら使用したレジスタを復帰して，割り込みから復帰する命令を実行するんだ．この命令を実行すると，退避していたプログラムカウンタとステータスレジスタをCPUが元に戻して，元のプログラムに戻るんだ.

## ● 初期化されていない静的変数の領域を0にクリアする

N先輩：C言語では初期化されていない静的変数の初期値は0だと規定されているので，初期化されていない静的変数は初期値をもたないとしてプログラミングすれば，0クリアしなくても良いんだけど，いちおう0クリアしておくべきだろうね.

C君　：じゃあ，スタック領域も0クリアしておけば，自動変数の初期値が0になって便利ですね.

N先輩：そりゃ，だめだよ．スタック領域は再利用されるから，最初は0でも，一度使われた後は値が書かれるからね.

C君　：あっ，そうですね.

N先輩：でもね，スタック領域を`0x55`とか，`0xaa`とかで初期化しておけば，デバッグ時にスタック領域をダンプしてどこまで使われたかを確かめることができるから，便利なこともあるんだ．スタック領域が足りなかったら，

- プログラムの暴走
- 予期しない変数の書き換え(図7)

といった現象になるから，それを見てスタック領域を増やしたりできるからね.

C君　：そうですね.

## ● 初期値をもつ静的変数の領域に初期値を転送する

N先輩：さっきいったように初期値はROMに置かれるから，そのままでは書き換えることができないんで，ROMの初期値をRAMに転送して書き換えができるようにするんだ.

C君　：初期値をもつ静的変数をプログラムで変更しなければROMからRAMに転送しなくても良いですよね.

N先輩：そうだね．その昔，CP/M-68KのCコンパイラでROM化してたときには，リンカの機能が貧弱だから，初期値をもつ静的変数をROMの領域に割り付けて，書き換えができない変数としてプログラミングしてたこともあったんだよ.

C君　：何とかなるものなんですね.

N先輩：でもね，その昔，初期値をもつ変数をROMに割り付

けるのに，メンバーに徹底されてなくて，その変数を書き換えるようなプログラムを書かれてしまって，デバッグ時に悩んだことがあるんだよ．変数のアドレスを見れば，ROMだとすぐに気づくんだけど，まさか，そんなことはしていないだろうというのが，頭のどこかにあって，ボードを交換してみたりして，ハードウェアを疑って，気づくのに時間がかかったことがあるんだ．

C君　：僕だったら，気づかないかもしれないですね．

N先輩：スタートアップルーチンで初期値を転送しなくても初期値をもった変数を定数として最初に初期値をもたない変数に代入する処理をアプリケーションに書くという方法もあるから，何とか書き換えできるようにもできるんだけどね．

C君　：あっ，そうですね．

N先輩：でも，本来，余分な処理だから，書かなくて良ければ，それに越したことはないんだけどね．スタートアップルーチンで初期化されていない静的変数領域を0にクリアして，初期値をもつ静的変数領域の初期値を転送してやれば，普通に意識せずにプログラミングできるからね（図8）．

C君　：そうですね．

## ハードウェアの初期化をするための準備

### ● ハードウェアの初期化

N先輩：ハードウェアの初期化は組み込み機器ごとに違うけど，
- CPU内蔵I/Oの初期化
- 出力ポートをOFF状態にする
- 各種LSIの初期化

なんかがあるよ．

C君　：ちょっと，イメージしにくいですね．

N先輩：組み込み機器ごとに違うから，説明もしにくいけど，ハードウェアに合わせた初期化が必要だってことだよ．

### ● ハードウェアのチェック

N先輩：ハードウェアのチェックはやり始めたらきりがない面もあるんだけど，組み込み機器の場合は最低限のチェックは必要だろうね．

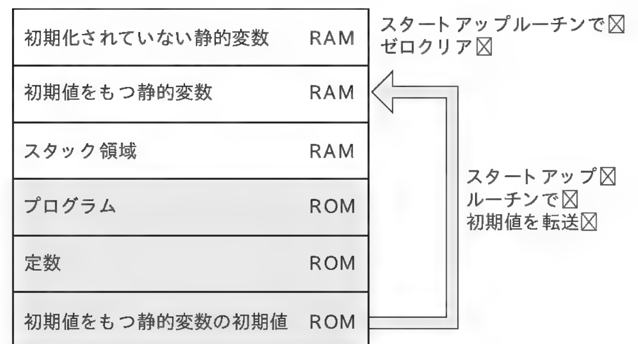
C君　：どんなチェックをするんですか．

N先輩：最低限必要なのは「RAMが正常に読み書きできるか」ってことかな．簡単に言うと，`0x55`と`0xaa`のように0と1のビットが反対の値を書いて，ちゃんと書けたかどうか確認するってな感じだけどね．

C君　：確かに，それをすると各ビットが0と1にできるってことになりますね．

N先輩：でもね，RAMチェックもけっこう奥が深くて，たとえばね，ハードウェアのバグですべてのアドレスが同じアドレスのRAMを指していたら，`0x55`と`0xaa`は両方とも書けるだろ．

〔図8〕スタートアップルーチンの処理




C君　：そりゃ，RAMだから，書けますね．

N先輩：だから，あるアドレスに書き込んで，他のアドレスが変化しないことを確認するとか，やり始めたらきりがなくなるんだけどね．

C君　：たいへんそうですね．

N先輩：でも，一度，作ってしまえば，他の機器でも同じようなことをすれば良いからね，

### ● 割り込み許可

N先輩：ハードウェアの準備が整ったら，割り込みを許可して，割り込みを受け付けるようにするんだ．

C君　：割り込みってもう一つ良くわからないんですが．

N先輩：割り込みは，よく電話にたとえられるけど，C君は仕事中に電話がかかってきたら，仕事を中断して電話に出るだろ．

C君　：はい，出ますね．

N先輩：この電話が割り込みみたいなものなんだよ．電話の内容によっては，電話が終わったら，すぐに元の仕事に戻ったり，電話の用件が急ぎだったら，その用件を先に処理して，元の仕事に戻ったりするだろ．

C君　：そうですね．

N先輩：割り込みも種類によって，すぐに済むものや，ある程度まとまった処理をする必要があるものがあるんだ．

C君　：割り込みも組み込み機器によって違うってことですね．

N先輩：そういうことだね．

### ● その他

N先輩：これら以外にも，たとえば，`malloc()`や`free()`などの動的なメモリ管理の機能を使う場合は，ヒープ領域を作っておくとか，組み込み機器の場合は環境変数はまず，使わないからやらないけど，そのプロセス用の環境変数領域を作成するとか，組み込み機器ごとに固有のものがいくつかあると思うよ．

C君　：そうなんですか．

## 組み込み機器用のCPUの条件

N先輩：Ｃ君はCPUといったら，何を思い浮かべる？

C君　：そりゃ，Pentiumプロセッサですよ．

N先輩：パソコンのCPUだったら，今だとPentiumプロセッサ系でクロックの高速なものってのが，そのパソコンの価値になるけど，組み込み機器の場合はCPUに何が使われていても製品価値はとくに変わらないんだ．

C君　：エアコンや冷蔵庫なんかの家電製品のCPUを気にしたことはないですからね．

N先輩：そうだね．組み込み機器用のCPUだったら，まず，必要な機能を満足できる安価なCPUといったことが，最初の条件となるんじゃないかな．だから，CPUクロックが高速で消費電力が多くて，発熱するからCPU用の冷却ファンが必要なCPUは，最初の段階で候補から外れてしまうんだよ．冷却ファンは，

- 価格アップにつながる
- 取り付け場所をどこにするか

といった点や機械的な駆動部分があるものは壊れやすいから，

- 冷却ファンが壊れたことの検出方法
- 冷却ファンが壊れたときの交換方法

といったことが信頼性の低下につながるんで，ネックになるんだ．

C君　：言われてみれば，そうですね．

N先輩：安価という意味はソフトウェアまで含めて考える必要があるから，

- 今まで使ったことがあるCPU
- 情報が簡単に入手できるCPU

といったことなんかも条件に入ることがあるんだ．

C君　：確かに，CPUについて調べるのも時間がかかりますからね．

N先輩：といっても，技術者としては，使ったことのないCPUを使ってみたくなるときもあるんだけどね．いまだと，SHシリーズやPowerPCなんかが候補になるだろうね．

## 組み込み機器のデバッグのしかた

### ● エミュレータを使ったデバッグ

N先輩：その昔はROM化システムのデバッグはCPUソケットにプローブを接続するICE（インサーキットエミュレータ）を使っていたものなんだ．ICEでのデバッグは，

- ターゲットCPU
- ターゲットメモリ（ROM/RAM）

などをICEの機能で差し替えて，

- 任意のアドレスでのブレーク（ソフトウェア/ハードウェア）
- リアルタイムトレース
- 実行時間測定

などの機能を利用してデバッグするんだ．この中でリアルタイムトレースが非常に便利でね．不具合が発生していそうなところの前後に設定しておけば，トレース結果をじっくり眺めることで微妙なタイミングのバグがよくわかったもんだよ．再現性の少ない不具合でも，一度，発生してくれれば良いから，便利なんだけど，トレースメモリに限界があるから，場所を絞り込むのがたいへんだったんだけどね．このタイプはCPUソケットに接続する必要があるから，CPUを取り外す必要があるけど，プローブの先端の形状とCPUソケットの形状が違うとソケットを変換したり，プローブのケーブルが短くてICEの取り付け位置に苦労したりしてけっこうめんどうだったんだよ．コンパイラの生成するシンボルを読めないと困るから，ICEがサポートしているシンボルに変換するツールを作成したこともあったしね．それに，最近は動作周波数の高いCPUの進化に追いついていけない状況になっているから，このタイプのICEは使われなくなってきているんじゃないかと思うよ．

C君　：そうなんですか．

N先輩：もう一つのタイプのICEとして
ROMソケットにプローブを接続するICEがあるんだ．このタイプのICEは，CPUはターゲットのものを使うから，CPUの動作周波数が高くても使える点がメリットなんだけど，ターゲット側にブレークポイントなどの制御プログラムが必要になるんだ．

C君　：リモートデバッガみたいですね．

N先輩：制御プログラムは小さいから，どうってことないんだけどね．最近のCPUにはJTAGインターフェースが用意されているものがあって，これとデバッガを接続する“JTAGデバッガ”が使われているみたいだよ．これもCPUの動作周波数が高くても使える点がメリットなんだけど，ユーザープログラムをターゲットメモリにダウンロードする必要があるので，ハードウェアがある程度完成していないと使えないという欠点があるんだ．

C君　：どれも，一長一短ってとこですね．

### ● ソフトウェアシミュレーションでデバッグ

N先輩：ICEは価格なんかも問題になったりして，ソフトウェアで，デバッグしてしまおうというのが，シミュレーションなんだ．

C君　：すごいですね．

N先輩：別にすごくなくて，パソコンやワークステーションな

んかでターゲットをシミュレーションするソフトウェアを作って，デバッグするんだけど，入出力をファイルで行ったりして，うまく作れば，良い線までいくんだけど，実際のI/Oやタイミングに関連したバグは見つけられないから，基本的にプログラムのロジックのデバッグ程度しかできないんだ．

C君　：それだけでも役に立つことはあるでしょうね．

N先輩：そうだね．

**● ターゲットに簡単な機能をもたせるリモートデバッグ**

N先輩：リモートデバッグはパソコンやワークステーションとターゲットの間をRS-232CやLANなんかで接続してデバッグする方法なんだ．ターゲット側に，

- 指定アドレスから指定バイト数の内容を返す
- 指定アドレスから指定バイト数だけ内容を変更する
- 指定アドレスでブレークする

といった比較的簡単な機能をもたせておくだけで良いんだ．その代わり，パソコン側のデバッガはこれらの機能を通信で利用して，ソースレベルのデバッグ機能をもたせないといけないから，かなりの規模のプログラムになってしまうんだ．

C君　：うちでは86系と68K系で使ってますよね．

N先輩：リモートデバッグもソフトウェアだけの機能だから，

- ハードウェアが完成していないと使えない
- ターゲットの資源を一部使う
- ハードウェアブレークが使えない

なんて多くの欠点があるけど，一度，移植すると複数の客先が重なっても，パソコンさえあれば，複数の人間が別々にデバックできるから便利なんだ．

C君　：うちは五月連休，お盆，お正月なんかはけっこうかちあいますからね．

## バグの原因の調査

**● ハードかソフトか――バグの発生の特定**

N先輩：組み込み機器だと，デバッグもちょっとめんどうなんだ．何か不具合が発生した場合，大まかに言うと，

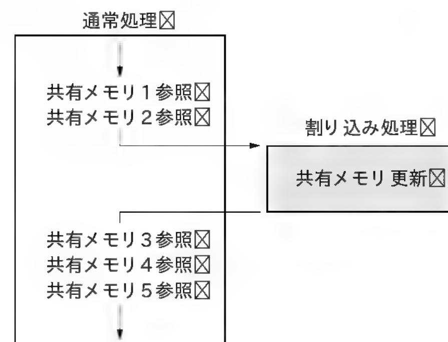
- ハードウェアの不具合
- ライブラリの不具合
- アプリケーションの不具合

なんかが考えられるよね．

C君　：そうですね．

N先輩：パソコン上のプログラムだと，バグの原因はほとんどアプリケーションのソフトウェアのほうで，たまに，ライブラリのバグってこともあるけど，組み込みプログラムの場合は，ハードウェアを新規に作ることも少なくないから，ハードウェアがらみのバグも少なくないんだ．

〔図9〕共有メモリの参照と更新




C君　：そうなんですか．

N先輩：ハードウェアを作成直後に，そのハードウェア用のBIOSを作成して，デバッグしているときは，プログラムが単純だから，再現性のあるバグとなりやすくて，ハードかソフトかの見極めもわりと簡単なんだけど，アプリケーションに組み込んだときに発生する不具合はハードかソフトかの見きわめが難しいこともあるんだ．

C君　：そういうもんですか．

N先輩：バグの特定は不具合の発生頻度が少ないほど難しいから，不具合が発生した状況を冷静に判断して，不具合が発生した状態にして，不具合を再現させるためのプログラムを作成したりもすることがあるんだ．一日に1回程度しか起きない不具合でも，その部分を繰り返し実行するテストプログラムを書けば不具合が発生する頻度を増やせるから，調べやすくなるからね．

C君　：不具合も発生しないと調べにくいですからね．

N先輩：いずれにしても，まず，最初はハードかソフトかを見きわめるためのテストプログラムを書いて，どちらに問題があるのか，切り分けることが必要なんだ．ハードに問題がありそうなときは，ハード屋さんといっしょに追いかけないといけないからね．といっても，ハードとソフトの両方に複数のバグが重なり合っていることもあるから，単純に切り分けができないことも少なくないんだけどね．

C君　：そういうときはたいへんでしょうね．

**● 割り込み処理と通常処理との排他制御**

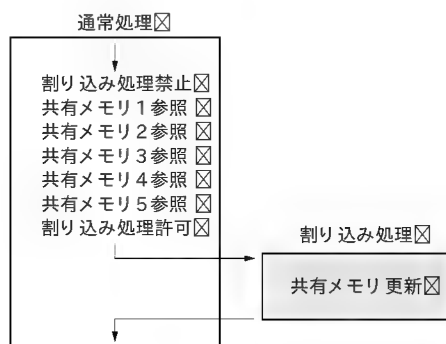
N先輩：ソフトウェアの不具合でも，割り込み処理と通常処理との排他制御に問題があると，けっこうたいへんなんだ．

C君　：どういうことですか．

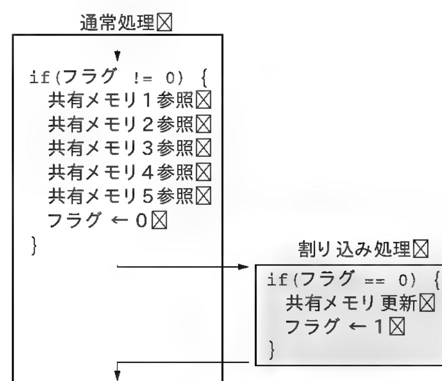
N先輩：たとえば，通常処理で共有メモリを参照している途中で，割り込みが発生して割り込み処理が共有メモリを更新してしまうと，通常処理に戻ったときに，途中から書き換えられた共有メモリを参照することになるだろ(図9)．

C君　：そうですね．

〔図10〕共有メモリの参照と更新（割り込み処理禁止）




〔図11〕共有メモリの参照と更新（フラグ制御）




N先輩：共有メモリが個々で無関係なら，それでも良い場合があるけど，共有メモリが関連しあっている場合は整合が取れなくなるんだ．

C君　：それで，そういうときは，どうすれば良いんですか．

N先輩：もっとも簡単なのは，通常処理で競合する割り込みが発生しないように，割り込みを禁止するんだ（図10）．

C君　：そうすれば，競合する個所で割り込みが発生しなくなりますね．

N先輩：でも，この方法は，割り込みを禁止する期間をできるだけ短くなるようにしないといけないんだ．だから，共有メモリを参照する間が短ければこれで良いんだけど，共有メモリが多いと割り込み禁止時間が長くなって，割り込みが待たされ過ぎて問題が発生することがあるんだ．

C君　：割り込みが長く待たされると割り込みの意味も薄れますからね．

N先輩：こういうときは，まともなリアルタイムOSを使っていれば，セマフォ制御などのAPIを使って，競合する個所をLock/Unlockで挟めば良いんだけど，そういうAPIがないときはフラグ変数で排他制御するんだ．割り込み処理で共有メモリに書き込んだら，フラグ変数をセットして，通常処理はフラグ変数がセットされていれば共有メモリを参照して，フラグ変数をリセットすれば良いんだ．割り込み処理はフラグ変数がリセットされているときにだけ，共有メモリを更新するようにすれば，排他制御できるんだ（図11）．割り込みが発生したときにフラグ変数がセットされていたら，通常処理の参照が終わった時点で，割り込みが呼び出されるようにしておけば良いんだ．

C君　：けっこうめんどうそうですね．

### ● 割り込みがらみのバグは難しい

N先輩：割り込みがらみのバグだと，追いかけるのもたいへんなんだ．リモートデバッガやICEなんかのデバッガを使ってブレークをかけると，それだけでタイミングが変わってしまうし，ブレークをかけてもプログラムは止まってるけど，周辺機器は動いているから，次の割り込みが発生したりして，通常の実行と状況が変わってくるから，不具合の再現ができないといったことも多いんだよ．

C君　：さっきの共有メモリのバグもタイミングによって起きたり起きなかったりしますからね．

N先輩：そういうことだね．割り込み処理と通常処理との排他制御がうまくいかなくて発生する誤動作は，割り込み処理と通常処理のタイミングがうまくいかないときにだけ発生して，たいていは正常に動作するから，不具合個所を見つけるのもたいへんなんだ．だから，設計時にしっかりと抑えておかないといけないんだよ．

C君　：今後，気をつけます．

## おわりに

本章では組み込みCプログラミングの基本について紹介しました．Windows上ではC言語単体での使用は少なくなりましたが，いろいろなCPU用のCコンパイラがある関係で，組み込み分野では，今後もC言語は多く使われていくものと思います．本章が組み込みCプログラミングの際の参考になれば幸いです．

参考文献

1) A.コーニング著，中村明訳，『Cプログラミングの落し穴』，（株）トッパン，1990年10月30日初版第3刷
2) B.W.カーニハン，D.M.リッチー著，石田晴久訳，『プログラミング言語C』第2版，共立出版（株），1989年6月15日初版1刷
3) 平林雅英，『ANSI C言語辞典』，技術評論社，平成元年10月25日初版第1刷
4) John H.Crawford+Patrick P.Gelsinger，岩谷宏訳，『80386プログラミング』，（株）工学社，昭和63年7月25日初版
5) インテルジャパン（株），i486マイクロプロセッサ プログラマーズ・リファレンス・マニュアル，1991年12月25日初版第1刷
6) Microsoft，Microsoft Visual C++ Version 5.0 Books Online
7) Microsoft，Microsoft Visual C++ Version 4.0 Books Online

8) Microsoft, Microsoft Visual C++ Version 2.0 Books Online
9) Microsoft, Microsoft Visual C++ Version 1.51 Books Online
10) Microsoft, Microsoft C 5.10 Optimizing Compiler User's Guide
11) Microsoft, Microsoft C 5.10 Optimizing Compiler Language Reference
12) Microsoft, Microsoft C 5.10 Optimizing Compiler Mixed Language Programming Guide
13) Microsoft, Microsoft C 5.10 Optimizing Compiler Run-Time Library Reference 1,2
14) Microsoft, Microsoft C 6.00A Professional Development System Programming Guide
15) Microsoft, Microsoft C 6.00A Professional Development System Reference
16) Microsoft, Microsoft C/C++ 7.0A Environment and Utilitiles
17) Microsoft, Microsoft C/C++ 7.0A Index/Error Messages
18) Microsoft, Microsoft C/C++ 7.0A Class Libraries Reference
19) Microsoft, Microsoft C/C++ 7.0A Class Libraries User's Guide
20) Microsoft, Microsoft C/C++ 7.0A C Language Reference
21) Microsoft, Microsoft C/C++ 7.0A C++ Language Reference
22) Microsoft, Microsoft C/C++ 7.0A C++ Programming Guide
23) Microsoft, Microsoft C/C++ 7.0A C++ Tutorial
24) Microsoft C/C++ ランタイム・ライブラリ・リファレンス Version 7, 監修マイクロソフト(株), 発行(株)アスキー
25) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 ユーザーズ ガイド
26) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 C++ チュートリアル
27) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 クラス ライブラリ ユーザーズ ガイド
28) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 プログラミング ガイド
29) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 Windows プログラミング ツールズ
30) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 プロフェッショナル ツールズ ユーザーズ ガイド
31) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 クラス ライブラリ リファレンス 1,2
32) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 ランゲージ リファレンス
33) Microsoft, Microsoft Visual C++ Development System for Windows Version 1.0 プログラマーズ リファレンス Vol.1～Vol.4
34) Microsoft Visual C++ ランタイム・ライブラリ・リファレンス Version 1, 監修マイクロソフト(株), 発行(株)アスキー, 1993年10月21日初版
35) Borland, Turbo C 2.0 User's Guide
36) Borland, Tuebo C 2.0 Reference Guide
37) Borland, Turbo C++ 1.0 Introduction
38) Borland, Turbo C++ 1.0 User's Guide
39) Borland, Turbo C++ 1.0 Programmer's Guide
40) Borland, Turbo C++ 1.0 Library Rerference Vol.1
41) Borland, Turbo C++ 1.0 Library Rerference Vol.2
42) Borland, Borland C++ 3.0 User's Guide
43) Borland, Borland C++ 3.0 Programmer's Guide
44) Borland, Borland C++ 3.0 Library Reference Vol.1
45) Borland, Borland C++ 3.0 Library Reference Vol.2
46) Borland, Borland C++ 3.0 Utilities Guide
47) Borland, Borland C++ 3.0 Quick Reference Guide
48) Borland, C++ Builder ユーザーズガイド
49) Borland, C++ Builder プログラマーズガイド
50) Borland, C++ Builder コンポーネント開発者ガイド
51) Borland, C++ Builder Visual Component Library リファレンス Vol.1, Vol.2
52) Borland, Borland C++ Builder 3 ユーザーズガイド
53) Borland, C++ Builder 3 開発者ガイド
54) C++ Builder 4 開発者ガイド, インプレス
55) C++ Builder 5 開発者ガイド, インプレス
56) C++ Builder 6 開発者ガイド, Borland
57) LSI C-80 Ver.3.4 ユーザーズマニュアル, エル・エス・アイ ジャパン(株)
58)『技術者のための Unix 系 OS 入門』, TECH I Vol.5, CQ出版(株), 2000年7月1日
59) 森友一朗, 薬師輝久, 馬場秀忠,『RTLinux リアルタイム処理プログラミングハンドブック』,(株)秀和システム, 2000年12月8日初版第1刷
60) Michael Beck, Harald Bohme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner 著,(株)クイック訳,『Linux カーネルインターナル』,(株)アスキー, 1999年9月21日初版第2刷
61) 奥脇学,『Linux プログラミングガイド』,(株)秀和システム, 2000年10月20日初版第1刷
62) 船木陸議,「RTLinux2.2系の導入と詳細」,『Interface』, 2001年6月号
63) 山本繁寿/有末一寿,「組み込み機器とプログラミング言語」,『Interface』, 2002年3月号
64) 山本繁寿/有末一寿,「組み込み機器とデバッグ環境」,『Interface』, 2002年3月号
65) 西田 亙,「オリジナルルートファイルシステムの構築」,『Interface』, 2002年7月号
66) セサミアン3人組 組込みソフトウェア管理者・技術者育成研究会,「長く活躍できる技術者になろう」,『DesignWave Magazine』, 2003年5月号
67) セサミアン3人組 組込みソフトウェア管理者・技術者育成研究会,「「組み込み」ならではの基礎知識」,『DesignWave Magazine』, 2003年5月号
68)『組み込み Linux 入門』, TECH I Vol.16, CQ出版(株), 2003年4月1日
69) ROM-Linux Install Guide ROM-Linux Ver0.94,(株)ワコムエンジニアリング
70) ROM-Linux Manual ROM-Linux Ver0.94,(株)ワコムエンジニアリング

なかしま・のぶゆき (株)Unix

オープンソースのITRON仕様OS

# TOPPERSで学ぶRTOS技術

## 第5回 サービスコールの概要・その2

岸田 昌巳

今回も引き続きμITRONのサービスコールについて解説を行います．

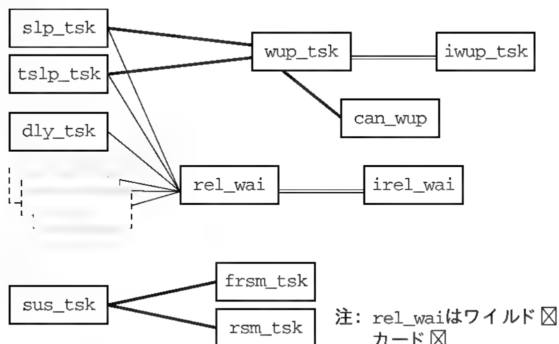
### タスク付属同期機能

TOPPERS/JSPでサポートしているタスク付属同期機能はslp_tsk，tslp_tsk，wup_tsk，iwup_tsk，can_wup，rel_wai，irel_wai，sus_tsk，rsm_tsk，frsm_tsk，dly_tskの11個があります．これらは図1のように対になっています．

### 起床待ち

| C言語API |
|---|
| ER slp_tsk(void);<br>ER tslp_tsk(TMO tmout); |
| パラメータ |
| TMO tmout; タイムアウト指定 |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS)，(E_NOSPT)，(E_RSFN)，(E_MACV)，(E_OACV)，(E_NOMEM)，E_CTX，E_PAR，E_RLWAI，E_TMOUT<br><br>E_CTX　：コンテキストエラー（待ち状態に入れない状態で呼び出した）<br>E_PAR　：パラメータエラー(待ち時間が不正)<br>E_RLWAI：待ち状態の強制解除（待ち状態の間にrel_waiを受け付けた）<br>E_TMOUT：ポーリング失敗またはタイムアウト(tslp_tskのみ) |

〔図1〕各サービスコールの関連




slp_tskのサービスコールを呼び出したタスク(自タスク)は起床要求があるまで待ち状態に入ります．この起床要求はキューイングされます．このため，slp_tskのサービスコールを呼び出す前に，wup_tskのサービスコールで起床要求が発行された場合，待ち状態に入りません．

また，TOPPERS/JSPではこのキューイングは1回しか行われません．これはJSPカーネルではwup_tskの起床要求はact_tskと同じく，1ビットのキューイング用バッファをTCB内に用意しているためです．

タイムアウトにはTMO_FEVR，TMO_POLが指定できます．

#### ● 返り値に関して(slp_tsk，tslp_tsk)

ディスパッチ保留の場合にはE_CTXが返ります．ディスパッチ保留状態は，タスク外から呼び出した場合や，CPUロック状態へ移行した場合が該当します．ディスパッチ保留状態では待ち状態に入ることができないため，slp_tskはエラーを返します．

tslp_tskを使用し，待ち時間の指定におかしな値を渡した場合，E_PARが返ります．

このタイムアウトを指定するパラメータの型はTMO型で，符号付き整数型です．

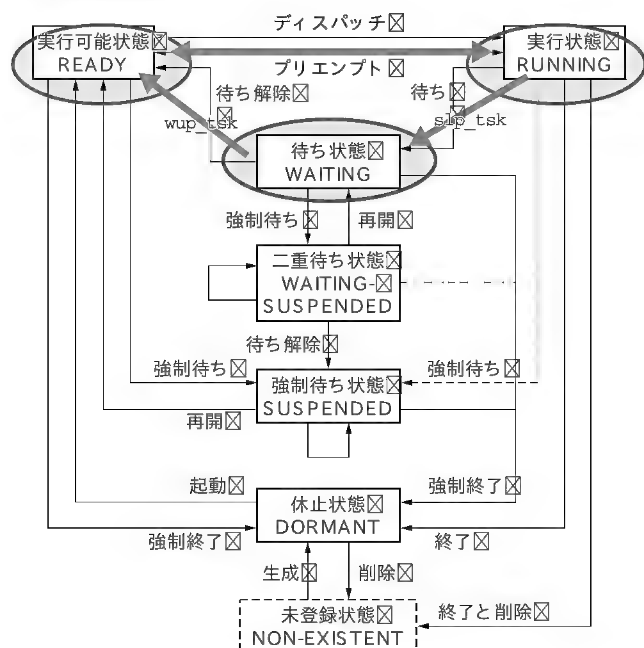
待ち時間の指定は正の値になりますが，ゼロとマイナス側をシステム側で使用しているので，ここでパラメータに使用できないTMO_NBLKを指定した場合や，システム側で使用していない負の値を指定した場合には，E_PARが返ります．

待ち状態の間にrel_waiを受け付けた場合は，E_RLWAIを返すことで，自タスクが待ち状態から強制解除されたことを示します．このため，強制解除を利用する場合は，slp_tskを利用する個所すべてに，E_RLWAIが返った場合の処理を入れる必要があります．

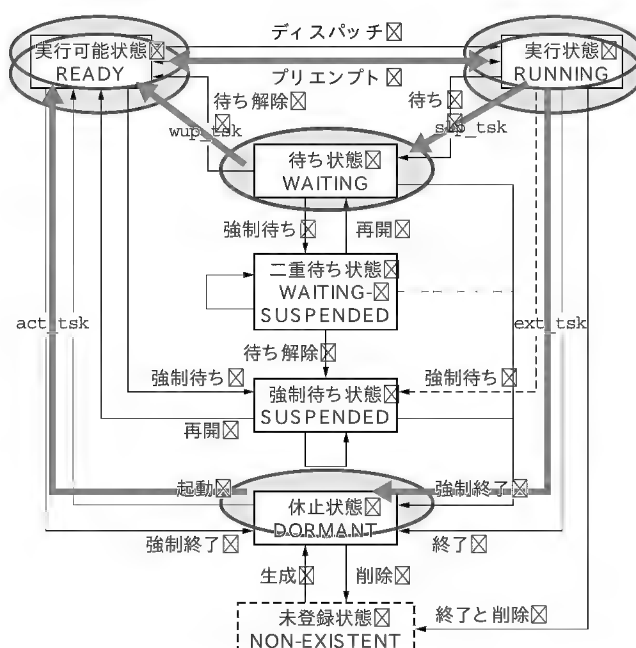
tslp_tskで指定した時間が過ぎた，またはポーリングに失敗した場合に，E_TMOUTが返ります．TOPPERS/JSPではTMO_POLを指定した場合にも，E_TMOUTが返ります．

なお，エラーコードのうち，TOPPERS/JSPで返さない値に関しては( )付きで表現しています．また，返ってくる来ないに関わらずエラー処理は必要と考えてください．

〔図2〕状態遷移図1




〔図3〕状態遷移図2




## タスクの起床

| C言語API |
|---|
| ER wup_tsk(ID tskid);<br>ER iwup_tsk(ID tskid); |
| パラメータ |
| ID tskid;　タスクID番号 |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, E_OBJ, E_QOVR<br><br>E_ID　：不正ID番号<br>E_OBJ　：オブジェクト状態エラー(対象タスクが休止状態)<br>E_QOVR：キューイングオーバフロー |

slp_tskのサービスコールで起床待ちに入ったタスク(起床待ち状態にあるタスク)の待ち状態を解除します．先に出てきた，slp_tskと組み合わせて使用します．

### ● 実際に使ってみる

slp_tsk，wup_tskのペアと，act_tsk，ext_tskのペアの違いは状態遷移図(図2，図3)で見るとわかりやすいでしょう．

休止状態から実行可能状態，実行状態に遷移するのと，待ち状態から実行可能状態，実行状態に遷移する違いがあります．slp_tsk，wup_tskのペアの場合，タスクが休止状態に入らないため，タスクの頭から実行されません．これはタスクの初期化処理が毎回実行されず，一度で済むことを示しています．たとえば，毎回初期化するとオーバヘッドになるプロトコルスタックなどの初期化処理では，一度の初期化で行うことができるため，スループットなどに影響が現れることになります(図4，リスト1)．

## タスクの起床要求を無効化

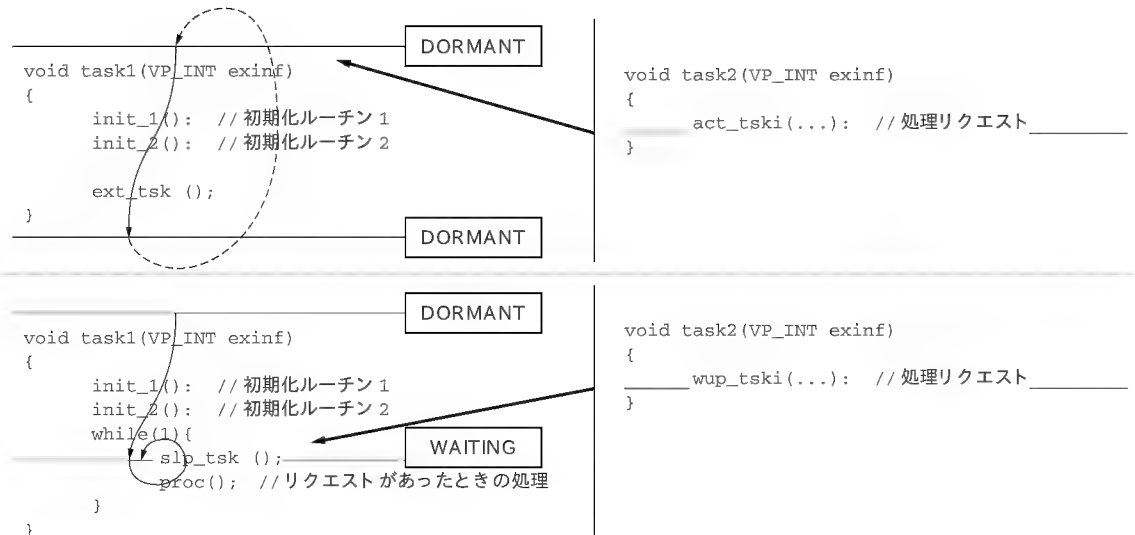
| C言語API |
|---|
| ER_UINT can_wup(ID tskid); |
| パラメータ |
| ID tskid;　タスクID番号 |
| リターンパラメータ |
| ER_UINT：キューイングされていた起床要求回数またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_OBJ |

指定したタスクへの起動要求をキャンセルします．このサービスコールは戻り値としてエラーコード，または起床要求がキューングされていた場合には，その回数を返します．

この回数で，タスクの処理が間に合っていない場合など，「どの程度間に合っていなかったか」を「何回起床要求が発行されていたか」を元に確認ができます．

たとえばリスト2では，一つ目のタスクは起床待ち(起床要求を受けるまで待ちつづける)，二つ目のタスクが起床要求を発行するとします．一つ目のタスクが起床した後，タスク内の処理を行い，処理が終わり再度起床待ちに入ろうとします．通常，処理が間に合っていれば，起床要求があるので，待ちに入らず，待ちから抜けてしまいます．ただし，ここではあえて1回分起床要求を溜めてタスク2に切り替わるようにしています．また，この例では，VBアプリケーションのボタンを押下することで割り込みが発生し，起床要求を発生させることができます．この起床要求が連続して発生され，起床した後から起床待ちまでの間に要求が複数「溜まってしまった」場合にキュー

〔図4〕実行時のパス比較図




〔リスト1〕タスクの起床のサンプル

```
void main_task(VP_INT exinf)
{
    init_log();              /*  ログの初期化  */
    init_loop_times();       /*  時間待ちループ回数の初期化  */
    while(1){
        wup_tsk(TASK_ID1);
        act_tsk(TASK_ID2);
       proc_something();

    }
}
/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "input task起動");
   while(1){
        slp_tsk();
         syslog_0(LOG_NOTICE, "slp_tskから抜けてきました");
    }
}

void output_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "output task起動");
   while(1){
        message("処理をはじめます");
        ext_tsk();          /* タスク終了(DORMANT状態へ) */
         syslog_0(LOG_NOTICE, "ここには着ません");
    }
}
```

〔リスト2〕タスクの起床要求を無効化する例

```
/*
 *  メインタスク
 */
void main_task(VP_INT exinf)
{
ER err;
ER_UINT count;

    init_log();                 /*  ログの初期化  */
    init_loop_times();          /*  時間待ちループ回数の初期化 */
    err = act_tsk(TASK_ID1);
    syslog_1(LOG_NOTICE, "input taskを起動 err = %d",error);

    syslog_0(LOG_NOTICE, "input task に制御を移す");
    dly_tsk(1); /* input task に制御を移す */

    syslog_0(LOG_NOTICE, "main task に制御が戻る");

    while(1){
        syslog_0(LOG_NOTICE, "input task  起床要求1 回目");
        wup_tsk(TASK_ID1);

        syslog_0(LOG_NOTICE, "起床要求を入れてください");
        wait_time(3);

        count = can_wup(TASK_ID1);
        syslog_1(LOG_NOTICE, "TASK_ID1の起床要求は %d です",
                                                      count);
        syslog_0(LOG_NOTICE, "main task 起床待ちに入る");
        slp_tsk();
    }
}

/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    error = iact_tsk(TASK_ID3); /* control タスクを起動 */
    syslog_0(LOG_NOTICE, "control タスクを起動");
}

/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "input task 起動");
    while(1){
        syslog_0(LOG_NOTICE, "input task 処理待ちにあります");
        slp_tsk();          /* タスクを待ちに入れる */
        syslog_0(LOG_NOTICE, "input task 待ちから抜けました");
        wup_tsk(MAIN_TASK);
    }
}


void output_task(VP_INT exinf)
{
// このサンプルでは output task には注目しません
//       この例題では本質的な意味はない
    syslog_0(LOG_NOTICE, "output task 起動");
    while(1){
        syslog_0(LOG_NOTICE, "処理をはじめます");
        ext_tsk();          /* タスク終了( DORMANT状態へ) */
        syslog_0(LOG_NOTICE, "ここには着ません");
    }
}

void control_task(VP_INT exinf)
{
    wup_tsk(TASK_ID1);/*  */
    syslog_0(LOG_NOTICE, "control_task 起床要求1 回追加");
}
```

イングされます．TOPPERS/JSPでは，このキューイングできる回数は1回分のみなので，何回要求があったとしても，can_wupの戻り値は1です．

### ● 補足事項

エラーコードと要求回数が同じ所から返って来るITRON仕様ではエラーコードはすべて負の数になっています．要求回数は0以上なので一つのリターンパラメータで兼ねています．

ただ，ここでは負の数であれば致命的な問題が発生しているわけで，また正の数であっても処理が追いついていないことが考えられるので，0以外は何らかの対策が必要だと思います．

## 待ち状態の強制解除

| C言語API |
|---|
| ER rel_wai(ID tskid);<br>ER irel_wai(ID tskid); |
| パラメータ |
| ID tskid;　タスクID番号 |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV),<br>(E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_OBJ<br><br>E_CTX : コンテキストエラー<br>(ディスパッチ禁止状態などで使用した) |

待ち状態にあるタスクを強制的に解除し，実行可能状態にします(図5)．対象となるタスクが，2重待ち状態にある場合は，強制待ち状態になります．

待ち状態に入るサービスコールとして表1に示すものがあります．これらに対してrel_waiは，待ち要因がまだ残っているのに待ち状態を解除し，実行可能状態にします(リスト3)．

rel_waiを使う場合，待ちに入るタスクの処理は，待ち要因を解除されても良い作りにしておく必要があります．

具体的には，待ちにサービスコールから返るエラーコードE_RLWAIを確認し，解除された要因に基づいて処理します．たとえば，処理が間に合わなかったので待ち要因を強制的に解除した場合などはリトライするか，エラー処理などを行います．これ以外にも，強制的にプログラムを終了するために待ちに入っているタスクを強制的に解除してタスクごとに終了処理を行わせることなどもあります．

### ● rel_waiを利用した場合のエラー処理

エラー処理はシステムを正常系/異常系に分類すると異常系にあたる処理が多く，設計やコーディング，検査が難しい部分でもあります．とくに待ち状態から強制的に解除し，エラー処理として何らかの処理を行うような場合，タスクの処理手順を決めておくことを勧めます．これは，解除後に何かを行う場合，該当する部分すべての足並みを揃えておかないと，漏れや抜けなどが発生しやすくなることから，処理の方法，コードの記述を揃えることで防止しようというものです．

たとえていうと，防災訓練のように，普段から準備しなければならない異常処理なのかもしれません．

### ● 強制待ち状態

強制待ち状態に移行する，強制待ち状態から再開するサービスコールにはsus_tsk，rsm_tsk，frsm_tskの三つがあります．

待ち状態と強制待ち状態の違いは，待ち状態が自タスクが呼び出したサービスコールで待ち状態に入るのに対し，強制待ち状態は，自タスクを含めた他のタスクから対象タスクを待ち状態に入れることができます．

待ち状態は待ち要因がなくなった時点で解除されますが，「強制待ち状態への移行」は，発行された回数ぶんだけネストし，

〔図5〕状態遷移図3



〔表1〕待ち状態に入るサービスコール

| サービスコール名 | 機能 |
|---|---|
| slp_tsk | 起床待ち |
| tslp_tsk | 起床待ち(タイムアウトあり) |
| dly_tsk | 自タスクの遅延 |
| wai_sem | セマフォ資源の獲得 |
| twai_sem | セマフォ資源の獲得(タイムアウトあり) |
| wai_flg | イベントフラグ待ち |
| twai_flg | イベントフラグ待ち(タイムアウトあり) |
| snd_dtq | データキューへの送信 |
| tsnd_dtq | データキューへの送信(タイムアウトあり) |
| rcv_dtq | データキューからの受信 |
| trcv_dtq | データキューからの受信(タイムアウトあり) |
| rcv_mbx | メールボックスからの受信 |
| trcv_mbx | メールボックスからの受信(タイムアウトあり) |
| get_mpg | 固定長メモリブロックの獲得 |
| tget_mpg | 固定長メモリブロックの獲得(タイムアウトあり) |

〔リスト 3〕待ち状態の強制解除の例

```
void main_task(VP_INT exinf)
{
ER err;
    init_log();             /*  ログの初期化  */
    init_loop_times();      /*  時間待ちループ回数の初期化  */
    syslog_0(LOG_NOTICE, "main task起動");

    /* タスク 1，タスク 2を起動します */
    err=act_tsk(TASK_ID1);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_ID1) %d",err);
    err=act_tsk(TASK_ID2);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_ID2) %d",err);

    /* タスク 1とタスク 2の優先順位を設定します */
    err=chg_pri(TASK_ID1,HIGH_PRIORITY);
   syslog_1(LOG_NOTICE, "chg_pri(TASK_ID1) %d",err);
    err=chg_pri(TASK_ID2,HIGH_PRIORITY);
   syslog_1(LOG_NOTICE, "chg_pri(TASK_ID2) %d",err);

    /* メインタスクの優先順位を下げます */
    err=chg_pri(MAIN_TASK,LOW_PRIORITY);
   syslog_1(LOG_NOTICE, "chg_pri(TASK_MAIN) %d",err);


    while(1){
       proc_something();

       syslog_0(LOG_NOTICE, "優先度の高いタスクに wup_tsk 発行");
       err=wup_tsk(TASK_ID1);
       syslog_1(LOG_NOTICE, "wup_tsk(TASK_ID1) %d",err);

       syslog_0(LOG_NOTICE, "優先度の低いタスクに wup_tsk 発行");
       err=chg_pri(TASK_ID1,LOW_PRIORITY);
       syslog_1(LOG_NOTICE, "chg_pri(TASK_ID1) %d",err);
       err=wup_tsk(TASK_ID1);
       syslog_1(LOG_NOTICE, "wup_tsk(TASK_ID1) %d",err);

       syslog_0(LOG_NOTICE,
                    "MAINタスクの優先順位を上げて wup_tsk 発行");

       err=chg_pri(MAIN_TASK,HIGH_PRIORITY);
       syslog_1(LOG_NOTICE, "chg_pri(TASK_MAIN) %d",err);

       err=wup_tsk(TASK_ID1);
       syslog_1(LOG_NOTICE, "wup_tsk(TASK_ID1) %d",err);
       err=wup_tsk(TASK_ID1);
       syslog_1(LOG_NOTICE, "wup_tsk(TASK_ID1) %d",err);
       err=wup_tsk(TASK_ID1);
       syslog_1(LOG_NOTICE, "wup_tsk(TASK_ID1) %d",err);

       syslog_0(LOG_NOTICE, "TASK_ID1タスクに can_wup 発行");
       err=can_wup(TASK_ID1);
       syslog_1(LOG_NOTICE, "can_wup(TASK_ID1) %d",err);

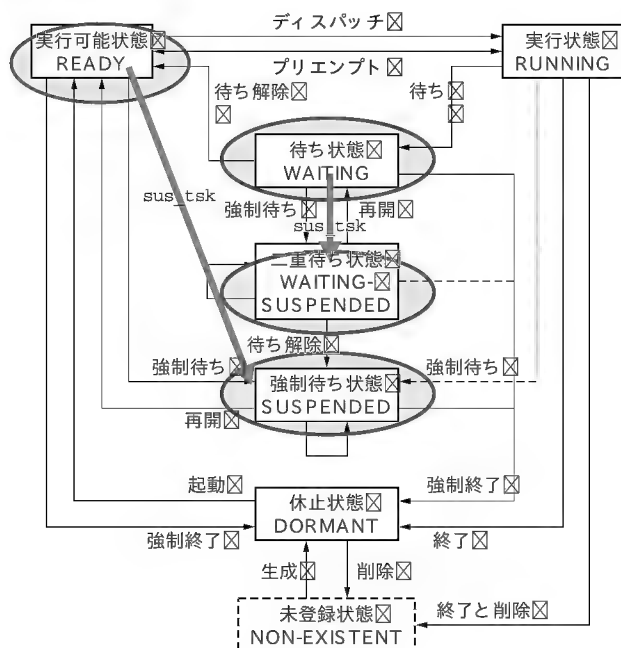
       syslog_0(LOG_NOTICE, "MAINタスクの優先順位を下げる");
       err=chg_pri(MAIN_TASK,LOW_PRIORITY);
       syslog_1(LOG_NOTICE, "chg_pri(TASK_MAIN) %d",err);

    }
}

/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "input task起動");
   while(1){
        slp_tsk();
        syslog_0(LOG_NOTICE, "input task処理実行");
    }
}

void output_task(VP_INT exinf)
{
    syslog_0(LOG_NOTICE, "output task起動");
   while(1){
        slp_tsk();
        syslog_0(LOG_NOTICE, "output task処理実行");
    }
}
```

〔図 6〕状態遷移図 4




要求を受け付けることができます．ネストとは入れ子状態のことです．このネスト状態は，発行された回数分だけ「強制待ち状態からの再開」を発行して元に戻さないと実行可能状態に入れません．

## 強制待ち状態への移行

| C言語API |
|---|
| ER sus_tsk(ID tskid); |
| パラメータ |
| ID tskid;　タスク ID 番号 |
| リターンパラメータ |
| ER E_OK（正常終了）またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_OBJ, E_QOVR |

指定したタスクを強制待ち状態にしてタスクの実行を中断させます（図 6）．TOPPERS/JSP では TCB 内にタスクの強制待ち要求ネスト数を管理する領域をタスクの状態を示すビットパターンで管理しています．このため，タスクの強制待ち要求ネスト数は最大 1になっています．よって，TOPPERS/JSP では実質的にネストしません．2回以上強制待ちを要求すると E_QOVR が返ります．

### ● 補足事項

JSP カーネルでは強制待ち状態のタスクの強制待ち要求ネスト数を管理する領域は，TCB 内にタスクの状態を示すビットと兼用で用意されています．ネスト可能数はこのビットで示すこ

〔リスト 4〕sus_tsk，rsm_tskの例

```
void main_task(VP_INT exinf)
{
ER err;
    init_log();              /*  ログの初期化  */
    init_loop_times();       /*  時間待ちループ回数の初期化  */

    /* タスク1，タスク2を起動します */
    err=act_tsk(TASK_ID1);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_ID1) %d",err);
    err=act_tsk(TASK_ID2);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_ID2) %d",err);

    /* タスク1とタスク2の優先順位を設定します */
    err=chg_pri(TASK_ID1,HIGH_PRIORITY);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_ID1) %d",err);
    err=chg_pri(TASK_ID2,HIGH_PRIORITY);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_ID2) %d",err);

    /* メインタスクの優先順位を下げます */
    err=chg_pri(MAIN_TASK,LOW_PRIORITY);
   syslog_1(LOG_NOTICE, "act_tsk(TASK_MAIN) %d",err);

    while(1){
        proc_something();
        err=rsm_tsk(TASK_ID1);
        syslog_1(LOG_NOTICE, "起こします  rsm_tsk(TASK_ID1) %d",
                                                          err);

        err=rsm_tsk(TASK_ID2);
        syslog_1(LOG_NOTICE, "起こします  rsm_tsk(TASK_ID2) %d",
                                                          err);
    }
}

/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
ER err;
    err=iact_tsk(TASK_ID3); /* 管理タスクを起動(READY状態へ) */
    syslog_1(LOG_NOTICE, "管理タスクを起動しました err=%d",err);
}

/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
ER err;
    syslog_0(LOG_NOTICE, "input task起動");
   while(1){
        syslog_0(LOG_NOTICE, "インプットタスク待ちに入ります");
        err=sus_tsk(TSK_SELF);
        syslog_1(LOG_NOTICE,
           "インプットタスク sus_tsk()から抜けました err=%d",err);
        syslog_0(LOG_NOTICE, "処理を行っています");
        wait_time(1);    /*  観察しやすくするための時間つぶし */
    }
}
```

とが可能な範囲で実装されています．このビットはタスクの状態を示すビットと兼用なので，実際にはネストした回数を覚えることができません．

ネストした回数を増やすための機能拡張するのであれば起床要求のサービスコールと同じようにビットフィールドで用意されている変数を単にビット拡張するだけではなく，別途カウンタなどが必要になります．

### 強制待ち状態からの再開

| C言語API |
|---|
| ER frsm_tsk(ID tskid); |
| パラメータ |
| ID tskid;　タスクID番号 |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_OBJ, E_QOVR |

sus_tskにて強制待ち状態に入ったタスクを元に戻します．あたりまえですが強制待ち状態にあるタスクは実行状態にないので，自タスクを指定することはできません．また，TOPPERS/JSPでは実質的にネストすることはありません．

### 強制待ち状態からの強制再開

| C言語API |
|---|
| ER rsm_tsk(ID tskid); |
| パラメータ |
| ID tskid;　タスクID番号 |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_OBJ, E_QOVR |

rsm_tskと同様に，sus_tskにて強制待ち状態に入ったタスクを元に戻します．

ちなみに，TOPPERS/JSPでは，frsm_tskとrsm_tskは同一の処理となっています．sus_tskを実際に使ってみるとリスト4のようになります．

### 自タスクの遅延

| C言語API |
|---|
| ER dly_tsk(RELTIM dlytim); |
| パラメータ |
| IRELTIM dlytim; 自タスクの遅延時間(相対時間) |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_PAR, E_RLWAI<br><br>E_PAR　：パラメータエラー(dlytimが不正)<br>E_RLWAI：待ち状態の強制解除(待ち状態の間にrel_wai受け付けた) |

指定時間自タスクを経過時間待ち状態にします．

● **dly_tskによるタスクの連携**

自タスクの時間待ちの間に他のタスクに処理を行ってもらおうという考え方で，タスク間の制御構造を作ることができます．ただし，必要な時間待ちを明確にした後で利用しないとタスク間の連携を明確に定義することはできません．

〔リスト 5〕dly_tskの例

```
/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
ER err;
    syslog_0(LOG_NOTICE, "input task起動");
   while(1){
        syslog_0(LOG_NOTICE, "dly_tsk待ちに入りました");
        err=dly_tsk(10);
        syslog_0(LOG_NOTICE, "処理を行っています");
    }
}
```

〔リスト 6〕DEF_TEXの使い方

```
DEF_TEX(TASK1, {TA_HLNG,tex_routine});
```

たとえば，周辺I/Oの状態変化をポーリングするのに確認間隔を空けるためなどの利用には良いのですが，他のタスクに処理にかかる時間だけ時間待ちを行い，処理を待つなどの利用方法は勧めません．

ソフトウェア部品の導入など，多数のタスクからなるシステムでは，逆にdly_tskを利用した安直な連携のやりかたでは問題を引き起こしかねません．具体的には，うまくタスク間の連携が取れない，性能の劣化が起こるなどの問題となります．このような設計では，スケーラブルなシステムにはなりえません．

プログラムの例示として，最初に出てくるサービスコールですが，タスク間の連携を図るためにはよく考えて利用する必要があります．タスクの連携には用いずに必要最小限の利用に留めることを勧めます．

dly_tskを実際に使ってみるとリスト 5のようになります．

## タスク例外処理機能

タスク例外処理はμITRON4.0で新たに導入された機能ですが，文字どおり例外処理を行うために提供されています．これは従来(μITRON3.0以前)は，実装依存として定義していなかった例外処理の枠組みを定義しています．

この例外処理の枠組みの中で，CPU例外ハンドラとタスク例外処理機能とを用意し，タスク例外処理機能ではタスクに行わせる例外処理を提供しています．このタスク例外処理はUNIXのシグナル処理を簡略化したような機能としてたとえられており，典型的な処理として以下を想定しています．

1) ゼロ除算などのCPU例外をタスクに伝える
2) 他のタスクに終了要求を出す
3) タスクにデッドラインが来たことを通知する

提供している機能自体はシンプルなものですが，より高度な例外処理を実現するための一部品として提供されているため，上記の用途に限定されるものではありません．

また，コーディング上のメリットとして，通常処理と例外処理とを分けることで処理内容を明確に，かつ簡潔に記述することができるようにもなります．

利用するにあたっては，静的APIにてコンフィギュレーションファイルに定義する必要があります．

**● DEF_TEX：タスク例外処理ルーチンの定義**

タスクごとに定義されるタスク例外処理ルーチンを定義します(リスト 6)．なお，スタンダードプロファイルの範囲(TOPPERS/JSP含む)ではタスク例外処理ルーチンの再定義はできません．

**● 登録の方法**

tex_routineという関数をTASK1のタスク例外処理ルーチンとして登録します．TA_HLNGは，この処理ルーチンが高級言語で記述されていることを意味しますが，TOPPERS/JSPではこれ以外の条件，アセンブラ言語での記述などはサポートしていません．記述時の指定はTA_HLNGだけとなります．

**● タスク例外処理ルーチンの入れ替え**

筆者の個人的な感想ですが，タスク例外処理ルーチンを入れ替えながら動作するアプリケーションタスクというものは，ほとんどないと思っています．もしあるとしても，実行環境によってタスクのふるまいが変わるなど，相当，限定したものになるのではないでしょうか．このため，スタンダードプロファイルの範囲で，再定義ができないことは大きな問題ではないと考えられます．

**● タスク例外処理の排他制御**

念のためですが，タスク例外処理は何時呼び出されるかわからないので，排他制御が必要なことはいうまでもありません．

たとえば，サービスコールを使う側の排他制御としては，登録したタスク例外処理自体の初期化，タスク例外処理を実行した後の後始末の間，必要になります．この間を再入可能にすることは難しく，再入可能にできなければ，再度タスク例外処理の要求が発生したときに，たいていは問題を引き起こします．このため，次に述べるタスク例外処理の禁止(dis_tex)，許可(ena_tex)を用います．

## タスク例外処理の要求

| C言語API |
|---|
| ER ras_tex(ID tskid,TEXPTN rasptn);<br>ER iras_tex(ID tskid,TEXPTN rasptn); |
| パラメータ |
| ID tskid; 要求対象タスクのID番号<br>TEXPTN rasptn; 要求するタスク例外処理のタスク例外要因 |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV),<br>(E_NOMEM), E_CTX, E_ID, (E_NOEXS), E_PAR, E_OBJ |

指定したタスクのタスク例外処理ルーチンを実行します．実行時の環境，おもにコンテキストとスタックですが，これは指定したタスクのものを使用します(図7)．

### タスク例外処理の禁止

| C言語API |
|---|
| ER dis_tex(void); |
| パラメータ |
| なし |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, E_OBJ |

自タスクをタスク例外処理禁止状態に移行させます．タスクが休止状態か，タスク例外処理ルーチンが定義されていない場合(スタンダードプロファイル以外では，NULLを定義した場合)はE_OBJが返ります．

このため，静的APIでタスク例外処理ルーチンを定義せず，dis_texを発行した場合にはエラーが返ります．

### タスク例外処理の許可

| C言語API |
|---|
| ER ena_tex(void); |
| パラメータ |
| なし |
| リターンパラメータ |
| ER E_OK(正常終了)またはエラーコード |
| エラーコード |
| (E_SYS), (E_NOSPT), (E_RSFN), (E_MACV), (E_OACV), (E_NOMEM), E_CTX, E_ID, E_OBJ |

自タスクをタスク例外処理許可状態に移行させます．

### タスク例外処理状態の参照

| C言語API |
|---|
| BOOL sns_tex(void); |
| パラメータ |
| なし |
| リターンパラメータ |
| BOOL TRUEまたはFALSE |

タスク例外処理禁止状態であるかどうかを返します．タスク例外処理禁止状態であればTRUE，許可状態であればFALSEを返します．

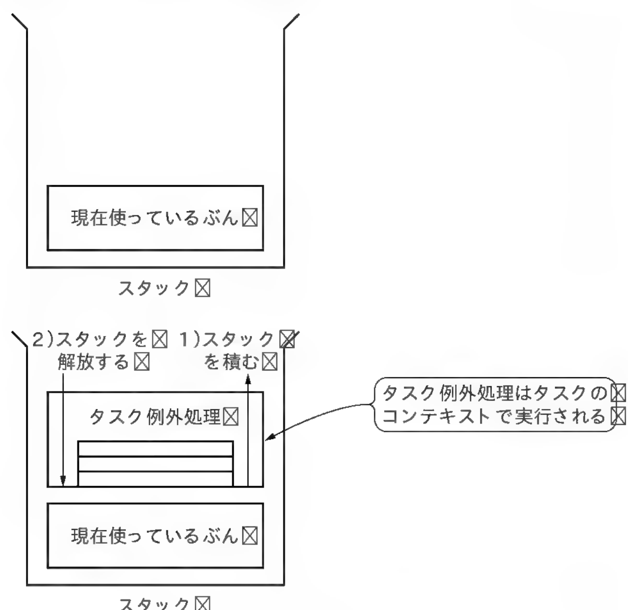
● 使い方

リスト7のようなコードを想定しています．このプログラムでは，“タスク例外処理が実行できないような処理を行っています”のメッセージに合わせて割り込みボタンを押してみてください．この間は割り込めないようにdis_texを発行しています．

### おわりに

前回とあわせてここまでで，タスクの管理機構と，タスクを直接操作する同期機構を説明しました．ここで説明したサービスコールは，静的APIが2個，C言語APIが23個でした．使いこなせそうな感触を持ってもらえたでしょうか．

〔図7〕スタックを使っている図




〔リスト7〕タスク例外処理状態の参照の例

```
/*
 *  起動されるタスク
 */
void input_task(VP_INT exinf)
{
ER err;
volatile UW i;

    syslog_0(LOG_NOTICE, "input task起動");
   while(1){
        syslog_0(LOG_NOTICE, "dly_tsk待ちに入りました");
        err=dly_tsk(1000);
        syslog_0(LOG_NOTICE, "処理を行っています");

        dis_tex();
        syslog_0(LOG_NOTICE, "割り込みボタンを押してタスク例外
                                    処理を要求してください");
        syslog_0(LOG_NOTICE, "dis_tex();が実行されているので
                                    あと10回は受け付けられません");
        for(i=0; i<10; i++){
            syslog_0(LOG_NOTICE, "タスク例外処理が実行できない
                                    ような処理を行っています");
        }
        ena_tex();
    }
}
```

次回からは，より複雑なシステムを構成するために必要となる同期・通信機能や，メモリプール管理機能などの説明に入ります．

内容として次回もTOPPERS/JSPのふるまいに関しても解説する予定ですが，ほとんどの解説部分はμITRON4.0仕様で共通の内容です．

きしだ・まさみ (株)フルノシステムズ

高速バス/大容量メモリとベクトル演算ユニットを活用した

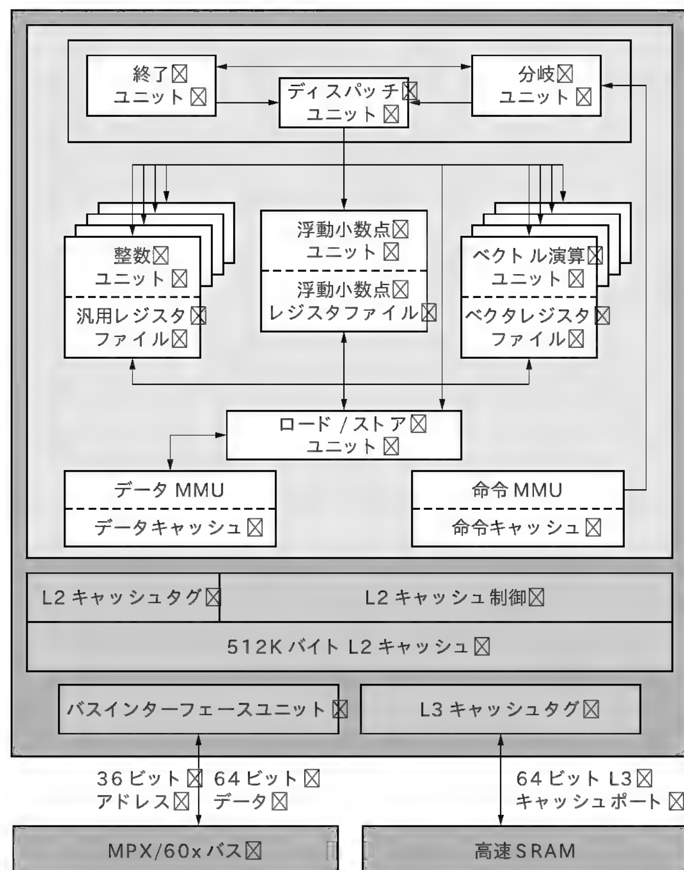
# PowerPC G4の概要とAltiVecを活かしたプログラミング技法

永野 和博

## はじめに

ディジタル信号処理を実現する方法として，一般的にはDSPを使ったり，場合によってはFPGAといった選択肢を思い浮かべる人もいるかもしれません．ここでは，ディジタル信号処理を実現する一つの解として，128ビットベクトル演算ユニットを搭載したPowerPC MPC74xxシリーズ(モトローラ)を紹介します．MPC74xxシリーズは，通称G4プロセッサと呼ばれ，アップルコンピュータのPowerMac G4(以下，PowerMac)にも採用されています．G4プロセッサは，ベクトル演算ユニットの性能を引き出すことで，マルチメディア処理に関してスーパーコンピュータに匹敵する性能を発揮してきました．今回はこのベクトル演算ユニットを活用して，プログラミング性能を大幅に向上させる方法を解説します．組み込み分野におけるディジタル信号処理を実現する手段として，またG4プロセッサを理解する一助となれば幸いです．

〔図1〕MPC7457のブロック図




## 1. PowerPCシリーズについて

### ● ファミリ展開

モトローラのPowerPCには，MPC60x/7xx/74xx/824xなど各種ファミリが存在します．これらPowerPCアーキテクチャをもつラインナップの中で，G4プロセッサ(MPC74xxシリーズ)は，ベクトル演算ユニットを搭載するハイエンドの製品として位置付けられています．このベクトル演算ユニットは通称AltiVecと呼ばれ，その実体はSIMD型の並列演算器から構成されます．

MPC74xxシリーズの中にはいくつかの品種があります．たとえば初期型のMPC7410は，命令の処理を4段パイプラインで行い，400MHzから500MHzまでの動作周波数で低コスト品が用意されています．最新のMPC7447/57は，命令の処理を7段パイプラインで行い，600MHzから1.3GHzまでの動作周波数ラインナップが用意されています．消費電力については，低電力版のMPC7447/57が1GHz動作時に10W以下を実現しています．

図1にMPC7457のブロック図を示します．MPC7457は，バックサイドのL3キャッシュインターフェースに，4MバイトまでのSRAMを接続することができます．MPC7447は，MPC7457のL3キャッシュインターフェースを省きパッケージを小さくした製品です．

### ● AltiVec

AltiVecは，32個の128ビットレジスタで構成されるベクタレジスタファイルと，四つのベクトル演算ユニットから構成され，これらは浮動小数点処理を行うユニット，積和演算などの複雑な演算処理を行うユニット，加算などのシンプルな演算処理を行うユニット，バイト単位の並べ替えを行うユニットに分

かれています．それぞれのユニットは，ベクタレジスタからデータを入力し，処理結果はベクタレジスタに格納します．

ディスパッチユニットからは，クロックごとに最大で3＋1（分岐ユニット）命令が発行されます．つまり，11個の演算ユニットの中から最大四つの演算ユニットが並列に動作します．ベクトル演算ユニットに対して命令が発行された場合，対象となる演算ユニットで128ビット処理が行われます．複数の演算ユニットを並列に動作させるためのスケジューリングについては，ハードウェアが行うため，ユーザーが意識しなくても並列動作が行われます．

## 2. G4プロセッサによる信号処理

信号処理を得意とするDSPの歴史は，汎用プロセッサに行わせていた処理の一部を切り出してDSPに行わせたことから始まり，汎用プロセッサとDSPは，それぞれの特徴を強化しながら進化してきました．しかし，半導体製造プロセスの進展にともなって機能の集積化が進み，汎用プロセッサとDSPの双方を隔てていた定義があいまいになってきています．ここでは，信号処理アプリケーションにおいても優位性を発揮するG4プロセッサの特徴を紹介します．

### ● プロセッサバスと高速SRAM用バス

通常DSPに搭載されているSRAMのサイズは数百Kバイト程度であり，処理データが内部SRAMに収まりきらない場合には，外部メモリにユーザーデータを展開することになります．キャッシュが搭載されていないDSPであればDMAを用いて内部SRAMと外部メモリ間の転送を行いながら細切れに処理することになります．

一方，G4プロセッサは，汎用のプロセッサとして発展してきたという歴史があるため，大規模なデータに効率良くアクセスする機能が強化されてきました．たとえばMPC7447/7457であれば，オンチップに大容量のキャッシュを搭載しつつ，メインの外部メモリに接続するプロセッサバスには最大167MHzで動作するMPXバスが用意されています．また，プロセッサバスとは独立に外付けのSRAMを接続することができる品種（MPC7410/7457）も用意されています．さらに，データストリームタッチ命令（データの先読み命令）を一つ発行するだけで，任意のデータブロックを外部メモリやL2/L3キャッシュからL1キャッシュにあらかじめ転送しておくことができます．この転送は，DSPで行われるDMAと同様に，コアの動作を妨げることなく行われます．表1では，他社のDSPとメモリアクセスの効率性を比較しています．この比較からも，G4プロセッサは，画像データなどの外部メモリに展開されている大きいデータブロックを扱う場合に，一般的なDSPよりデータアクセスの点で優位に立てることがわかります．

### ● 高いCPUコア性能

次にCPUコアの処理性能について説明します．信号処理アプリケーションにおいては，使用方法に合わせた処理性能を表すため一般的なMIPSやFLOPSではなく，単位時間あたりの積和演算性能で比較するほうが現実的だと考えます．表2では，単位時間当たりの積和演算性能について，浮動小数点処理をサポートしていて同等の価格帯にある他社のDSPと比較しています．

G4プロセッサに搭載されているAltiVecは，128ビットで演算可能なSIMDアーキテクチャを採用しています．そのため，単精度浮動小数点演算の場合，1クロックで四つの演算処理を実行することが可能です．たとえば，600MHzのMPC7447であれば，単精度浮動小数点形式で1秒間に24億回の積和演算（2400MMAC）を実行することができます．さらに，16ビットの整数形式/固定小数点形式で，1秒間に48億回の積和演算（4800MMAC）を実行することができます．

〔表1〕メモリアクセスの効率性比較

| | A社DSP | B社DSP | MPC7410 | MPC7447 |
|---|---|---|---|---|
| クロック（MHz） | 225 | 300 | 500 | 733 |
| 外部バスアクセス幅（ビット） | 32 | 64 | 64 | 64 |
| MAXバス周波数（MHz） | 113 | データなし | 133 | 167 |
| 転送レート（Mバイト/秒） | データなし | 800 | 1024 | 1336 |
| L1キャッシュ/L1メモリ（Kバイト） | 4（命令）＋4（データ） | 256（命令）＋512（データ） | 32（命令）＋32（データ） | 32（命令）＋32（データ） |
| L2キャッシュ/L2メモリ（Kバイト） | 256 | — | 2048（外付け） | 512 |

〔表2〕コアの演算処理性能比較

| | A社DSP | B社DSP | MPC7410 | MPC7447 |
|---|---|---|---|---|
| クロック（MHz） | 225 | 300 | 500 | 733 |
| サポートされている演算形式 | 整数形式/固定小数点形式/浮動小数点形式 | 整数形式/固定小数点形式/浮動小数点形式 | 整数形式/固定小数点形式/浮動小数点形式 | 整数形式/固定小数点形式/浮動小数点形式 |
| 16ビット整数形式/固定小数点形式による積和演算〔M（メガ）MAC/秒〕 | 450 | 2,400 | 4000（AltiVec）/500（整数ユニット） | 5864（AltiVec）/733（整数ユニット） |
| 32ビット浮動小数点形式による積和演算〔M（メガ）MAC/秒〕 | 450 | 600 | 2000（AltiVec）/500（浮動小数点ユニット） | 2932（AltiVec）/733（浮動小数点ユニット） |
| 64ビット浮動小数点形式による積和演算〔M（メガ）MAC/秒〕 | ハードウェアのサポートなし | ハードウェアのサポートなし | 500（浮動小数点ユニット） | 733（浮動小数点ユニット） |

## 3. 開発環境例

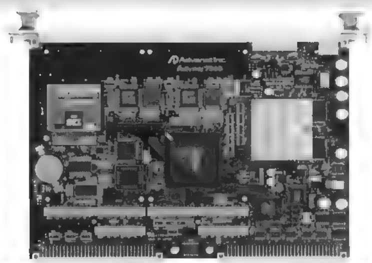
G4プロセッサが搭載されている評価ボードは，モトローラおよびサードパーティ各社から購入することが可能です(コラム参照)．また，G4プロセッサが搭載されているPowerMacを入手してAltiVecの評価を行うことができます．ここで紹介するAltiVecを使用したサンプルプログラムについては，開発にはPowerMacを，厳密な性能評価には評価ボードを使用しました．

PowerPC用の開発ツールは，多くのサードパーティからリリースされています．ここで，AltiVecの機能評価を行う場合，PowerMacを入手すれば，アップルのWebサイトから開発ツール(Project Builder)をダウンロードしてAltiVecの評価を開始することができます．また，メトロワークスより販売されているCodeWarrior for Mac OS Xを入手すれば，高いコンパイル速度をもつ開発環境でAltiVecの評価が可能です．

開発例を挙げると，最終製品のOSにLinuxを使う場合であれば，初めに，G4プロセッサが搭載されているPowerMacとYellow Dog Linuxの環境でユーザーアプリケーションを作成します．ネイティブ環境で開発を行えば，AltiVecプログラミングにかける期間を短縮できます．そしてG4プロセッサが搭載された評価ボード上でLinuxを走らせることで，最終製品に近い構成でデバイスドライバの用意と性能評価を行えます．最後に，完成品に近いソフトウェアを，開発したハードウェアに実装することで，開発初期段階のコストを抑えつつ，リスクの少ない迅速な開発が可能になります．

### G4プロセッサ搭載ボード　Column

ここではG4プロセッサが搭載されている，サードパーティ製のボードとして，A6PCI7504/Advme7505(アドバネット)を紹介します．A6PCI7504/Advme7505(写真A)は，PowerPC G3またはG4を搭載した6Uサイズ1スロット幅のCompactPCI仕様またはVME仕様のCPUボードです．1Mバイトあるいは2MバイトのL2キャッシュ，128M～512Mバイトのメインメモリ，ブートROMとして512Kバイトのフラッシュメモリ，システムコントローラにはGT-64260，10/100Base-TX EthernetおよびシリアルポートCompactFlashスロットなどを搭載しています．(編集部)



〔写真A〕Advme7505の外観

## 4. AltiVecプログラミング

● AltiVecプログラミングの基本

AltiVecを使用するためには，専用の命令を使う必要があります．しかしユーザーは，組み込み用の関数を用いてC言語で開発を行うことができます．組み込み関数に関してのマニュアル(AltiVec Technology Programming Interface Manual)は，以下のサイトからダウンロードすることができます．

`http://e-www.motorola.com/files/32bit/doc/ref_manual/ALTIVECPIM_D.pdf`

● 絶対値に変換する

それでは，C言語でAltiVecを使う際の基本的な流れについて紹介します．まずはもっとも簡単なAltiVec使用例として，メモリの値を絶対値に変換するプログラムを示します．

最初に，ベクタロード命令(`vec_ld`)を用いてメモリのデータをベクタ型で定義した変数へ格納します(図2)．このベクタロード命令はハードウェアの動作としては，16バイト分のデータを16バイト境界にアライメントされているアドレスからベクタレジスタに転送します．メモリ領域を16バイト境界から確保するには，ベクタ型で配列を定義したり`malloc`を用いるなどの方法があります．

次に，このデータに対して処理を行います．ここでは，各データの絶対値を取得する命令(`vec_abs`)を使います(図3)．処理が済んだデータは，ベクタストア命令(`vec_st`)を用いてメモリにストアします(図4)．このベクタストア命令はハードウェアとしては，16バイト分のデータをベクタレジスタから16バイト境界にアライメントされているアドレスへ転送します．

以上の一連の処理のプログラムを，リスト1に示します．また，実際の処理速度は変わりませんが，リスト1(b)のように省略した記述も可能です．

● バイトの並び替え

次にバイト単位の並べ替えを1サイクルで実行するパミュートユニットの機能を紹介します．パミュートユニットにより処理されるベクタパミュート命令は，制御用のベクタ型データをインデックスにして，二つのベクタレジスタの内容から指定されたバイトを選択して並べ替えを実行します(図5)．

パミュートユニットは，ベクタパミュート命令以外に，データの複製，パック/アンパックなどの命令処理を司ります．パミュートユニットによる並べ替え機能の効果的な応用例として，RGB→YCrCbの色変換処理に使っているサンプルプログラムを，次のサイトからダウンロード可能です．

`http://e-www.motorola.com/webapp/sps/site/`

overview.jsp?nodeId=03C1TR0467mKqW5Nf2F9
DHMbVXVDcM

● 4行4列の行列乗算

次はもう少し高度な処理例として，3次元画像処理にも使用される行列の乗算を用いて，AltiVecプログラミングの応用例を紹介します．ここでは4行4列の要素をもつ行列同士の乗算を実装してみます．行列の各要素は，浮動小数点形式とします．

PowerMacを用いて作成したプログラムのおもなファイル構成を以下に示します．

- MACOS_tst_matrix_mult_4x4.c
  …メインルーチン
- matrix_mult_4x4.c
  …行列乗算部
- vec_matrix_mult_4x4.c
  …AltiVecで最適化された行列乗算部
- asmfunc.c
  …初期化と計測用のサブルーチン(CodeWarrior用)
- matrixmultiply.mcp
  …CodeWarrior用プロジェクトファイル
- Pbsample.pbproj
  …Project Builder用プロジェクトファイル(XCODEでも利用可)

〔リスト1〕絶対値に変換するプログラム

```
get_abs(float* pfInput, float* pfOutput)
{
    vector float vfDataIn,vfDataOut;
    vfDataIn = vec_ld(0,pfInput);
    vf vfDataOut = vec_abs(vfDataIn);
    vec_st(vfDataOut,0,pfOutput);
}
```

(a) 基本形

```
get_abs(float* pfInput, float* pfOutput)
{
    vec_st(vec_abs(vec_ld(0,pfInput)),0,pfOutput);
}
```

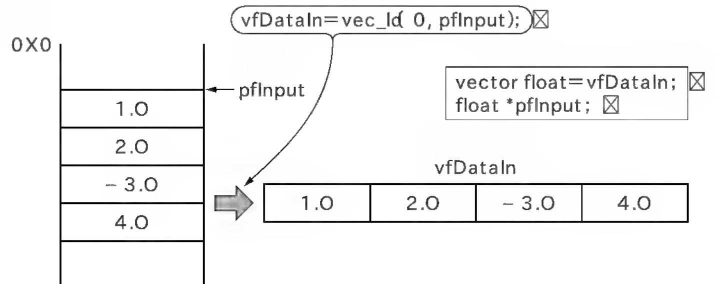
(b) 省略形

AltiVecによるインプリメントに際して注意した点は，次のとおりです．

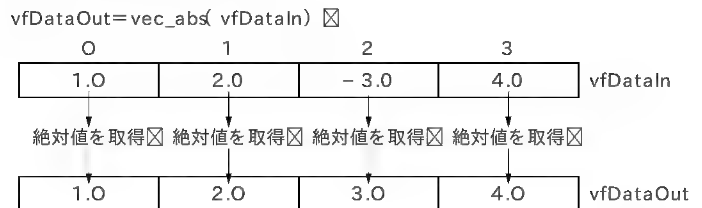
(1) AltiVecを使用する際にアセンブリ言語は用いず，組み込み関数でインプリメント

(2) メモリアクセスが連続的なアドレスで行われるアルゴリズムを考慮

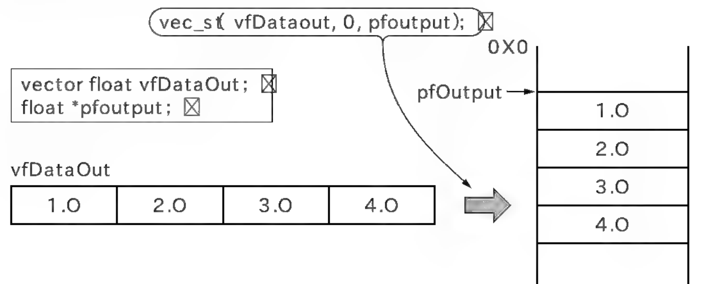
〔図2〕ベクタロード命令の動作



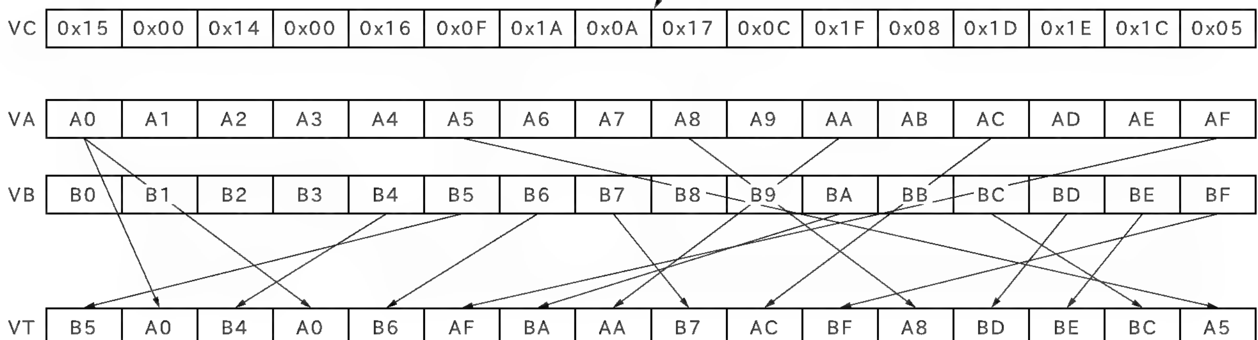

〔図3〕vec_abs命令の動作



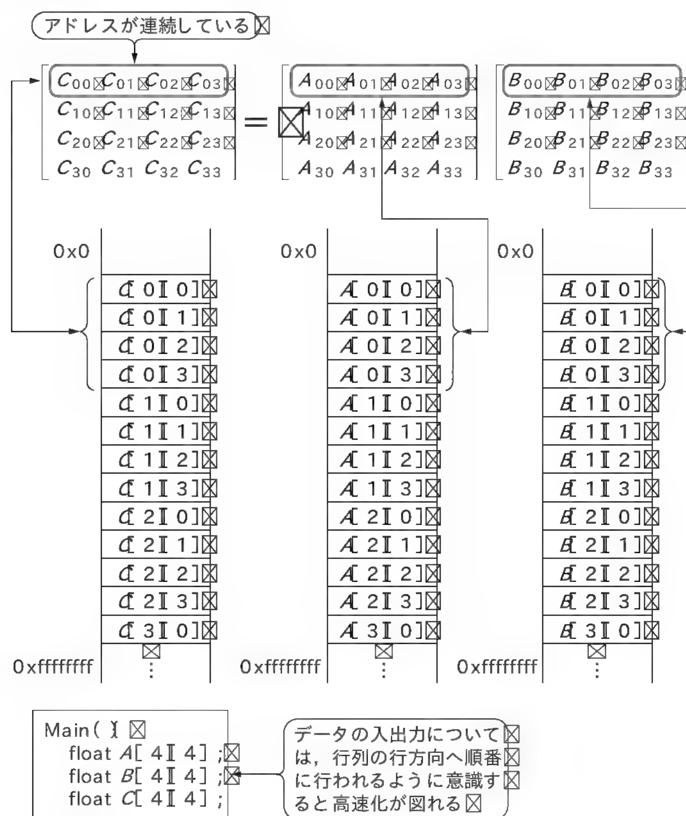

〔図4〕ベクタストア命令の動作



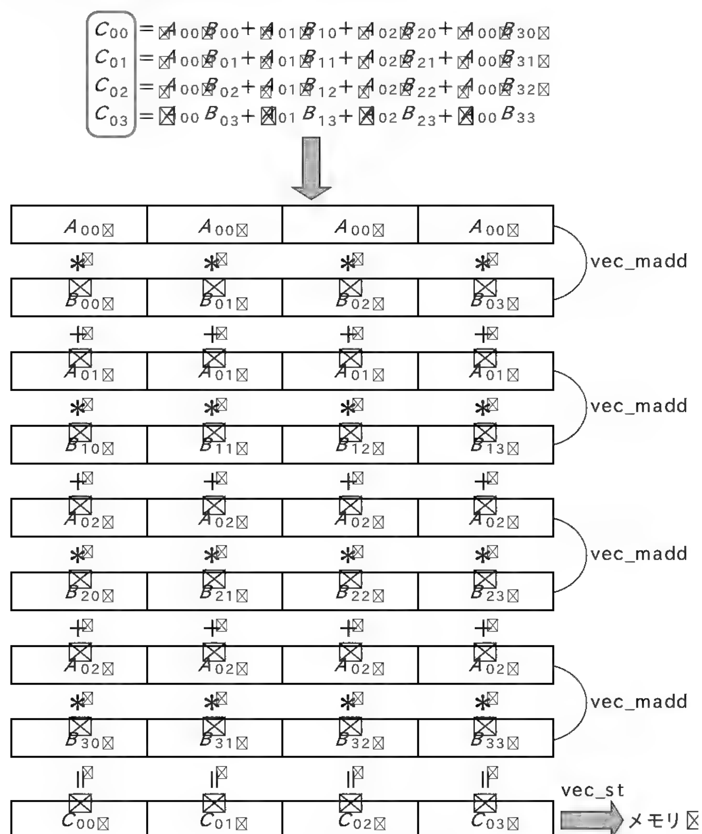

〔図5〕ベクタパミュート命令の動作



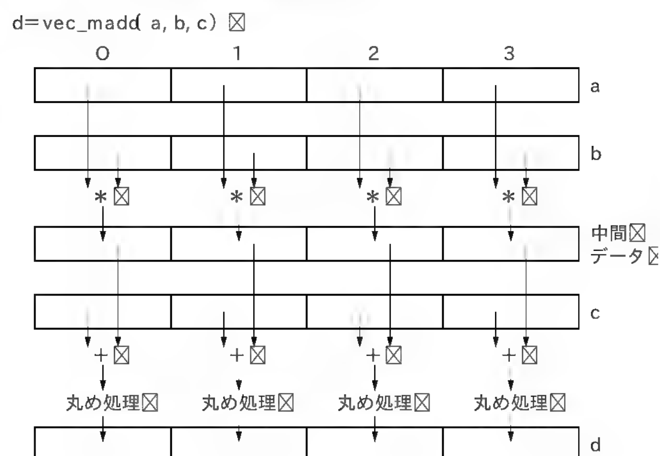
〔図6〕行列要素が格納されているイメージ




〔図8〕行列乗算の並列処理（浮動小数点形式）




〔図7〕vec_madd命令（積和演算）の動作




（3）AltiVecの並列処理命令がうまく適用されるアルゴリズムを考慮

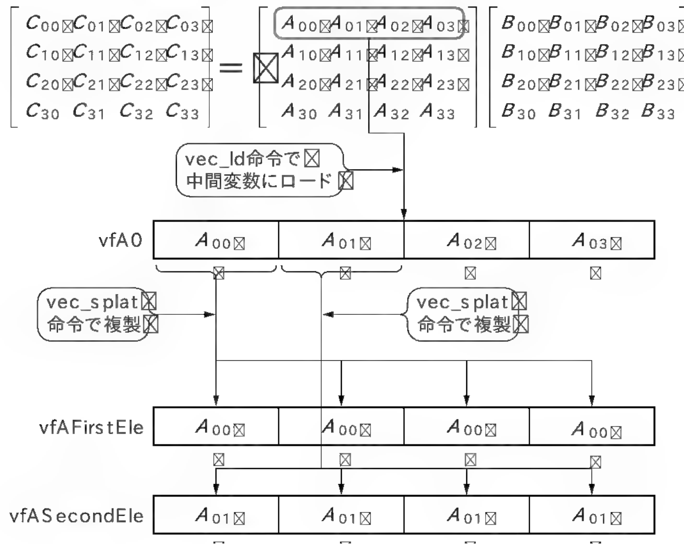
はじめに（1）についてですが，すべてをアセンブリ言語で記述するAltiVecプログラミングは，性能を極限まで引き出すことができますが，プログラムの容易性やコードの可読性が犠牲になります．そこでAltiVec用の組み込み関数を用いてC言語によりプログラミングすることで，プログラミングの容易性やコードの可読性を確保しつつ高い性能を引き出すことができます．今回はアセンブリ言語を使わずにAltiVec用の組み込み関数を用いて最適化を行います．

次に（2）についてですが，AltiVecによる最適化プログラミングを行うときには，ベクタロード命令，ベクタストア命令を中心にアルゴリズムを考えます．今回の場合は，まず処理対象データが配置されている並びは図6のようになるため，行要素にまとめてアクセスすると，メモリに対して連続的にアクセスできることがわかります．連続的にアクセスできれば，ベクタロード命令，およびベクタストア命令による16バイト転送命令が適用可能になり，プログラムの高速化が図れます．

最後に（3）についてですが，まずサポートされているAltiVec命令を調べます．今回は浮動小数点形式の積和演算を行うvec_madd命令が使えそうです（図7）．このvec_madd命令を実際に使うには，複数のデータをまとめて処理したほうが効率が良さそうです．そこで，出力結果となる行列Cの行要素をまとめて処理することを考えたのが，図8の処理順序です．図8の行列Aの要素は，vec_ld命令でメモリからロードしたデータをvec_splat命令で並べ替えて用意します（図9）．同じく行列Bの要素については，vec_ld命令でロードした値をそのままvec_madd命令に使用します．

この手順を繰り返すことで，出力となる行列Cの1行分が算出されます．この結果をvec_st命令でメモリにストアして1行分の処理が完了します．リスト2が1行分の処理プログラム

〔図9〕vec_splat命令によるデータの並べ替え




になります．

この手順を4行分繰り返すことで，すべての結果が算出されます．完成したプログラムを用いて，MPC7455(MPC7457の前機種)の評価ボードで計測した結果，1回の行列乗算あたりにかかるCPUサイクル数は41サイクルでした．このサイクル数には，データをキャッシュからロードしてから，処理結果をキャッシュにストアするまでの処理時間が含まれます．サイクル数計測の詳細については次項で解説します．

● 繰り返し処理の最適化

さきほどのプログラムの繰り返し処理は，呼び出し側(メイン関数)でループを用いて行っているため，ループの繰り返しごとに関数呼び出しに伴うオーバヘッドが加わります．このオーバヘッドを減らすための対策として，次の二つの方法があります．

(1) 行列演算を行っているファイルの中にループ処理を移す

(2) コンパイラのインライン展開オプションを用いる

(1)の方法は，行列乗算関数の引き数に，ループの繰り返し回数を加えて，行列演算が記述されているファイルへループ処理を移すことで対応します．この方法のメリットは，繰り返し実行される場合に，プログラムサイズを増大させずに関数呼び出しのオーバヘッドを削減することができる点です．

それに対して(2)の方法は，マクロのように展開されるため関数が呼ばれるごとにプログラムメモリを消費することになりますが，(1)よりもさらに関数呼び出しにともなうオーバヘッドを削減することが可能になります．これら二つの方法は，状況に応じて使い分けてください．

インライン展開を用いる場合は，次に示すとおり，行列乗算部のファイル形式をヘッダファイル(拡張子.h)にしています．

● inlined_matrix_mult_4x4.h

〔リスト2〕行列1行分の処理(浮動小数点形式)

```
vfAFirstEle = vec_splat(vfA0,0);
vfC0 = vec_madd(vfAFirstEle, vfB0,(vector float)(0));
vfASecondEle = vec_splat(vfA0,1);
vfC0 = vec_madd(vfASecondEle, vfB1,vfC0);
vfAThirdEle = vec_splat(vfA0,2);
vfC0 = vec_madd(vfAThirdEle, vfB2,vfC0);
vfAFourthEle = vec_splat(vfA0,3);
vfC0 = vec_madd(vfAFourthEle, vfB3,vfC0);
vec_st( vfC0, 0, out);
out += 4; //出力ポインタの更新
```

…インライン展開される行列乗算部

このインライン展開による高速化手法を用いた場合，1回の行列乗算あたりにかかるCPUサイクル数が41サイクルから30サイクルに高速化されました．

● 16ビット整数形式によるインプリメント

4行4列の行列乗算処理に，さらなる最適化を考えます．入力の行列要素が16ビット整数形式で，出力の行列要素が32ビット整数形式で処理する場合の処理例を示します．

浮動小数点形式の場合，一つのベクタレジスタに格納できるデータは四つまでですが，16ビット長のデータ形式であれば一つのベクタレジスタに八つまで格納することが可能になります．そのため，入力データの精度が16ビット長で演算できるのであれば，八つのデータをまとめて処理できるため，さらなる高速化が可能になります．

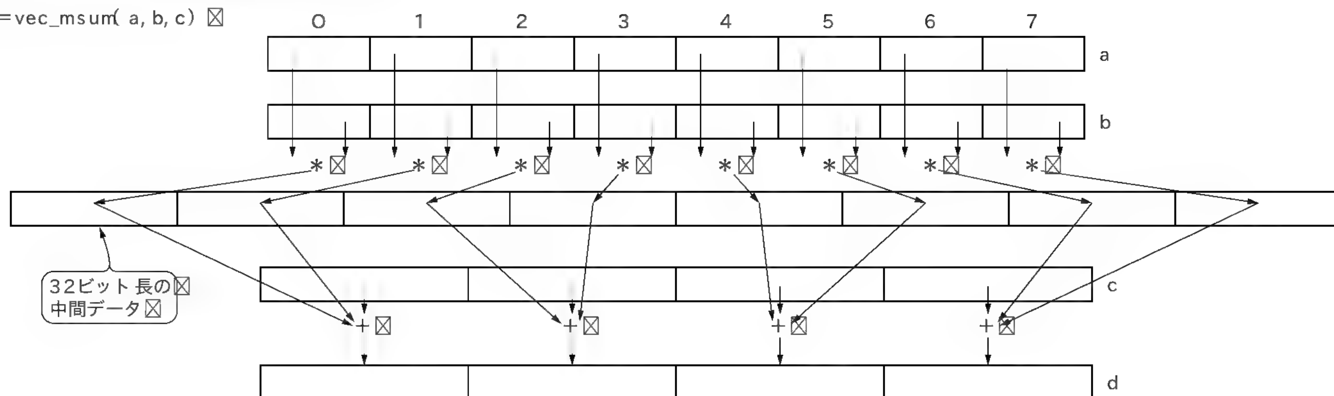
アルゴリズムを考える場合，浮動小数点形式の時と同様に，連続的なメモリアクセスになるように考慮します．出力結果が32ビットになるため，今回も行列Cの1行分をまとめて処理する方法を考えます．次に，入力が16ビット整数で演算結果が32ビット整数になるAltiVec命令を探します．これにはvec_msum命令が使えそうです．図10にvec_msum命令の動作を示します．この命令を使うと，図11でくくられている8個の演算を一度に処理できないでしょうか．

そこで，このくくられている処理を先程のvec_msum命令に当てはめて考えてみます(図12)．次に，このvec_msum命令で処理するために必要な入力データの並びを，ベクタパミュート命令で実現します．図13に示す手順で，ベクタ型の中間変数にロードしてきた入力データをベクタパミュート命令(vec_perm)で並べ替えます．ここで，ベクタパミュート命令に使用される並べ替え用のインデックスは，あらかじめベクタ型で定義した変数で用意しておきます(リスト3，p.137)．

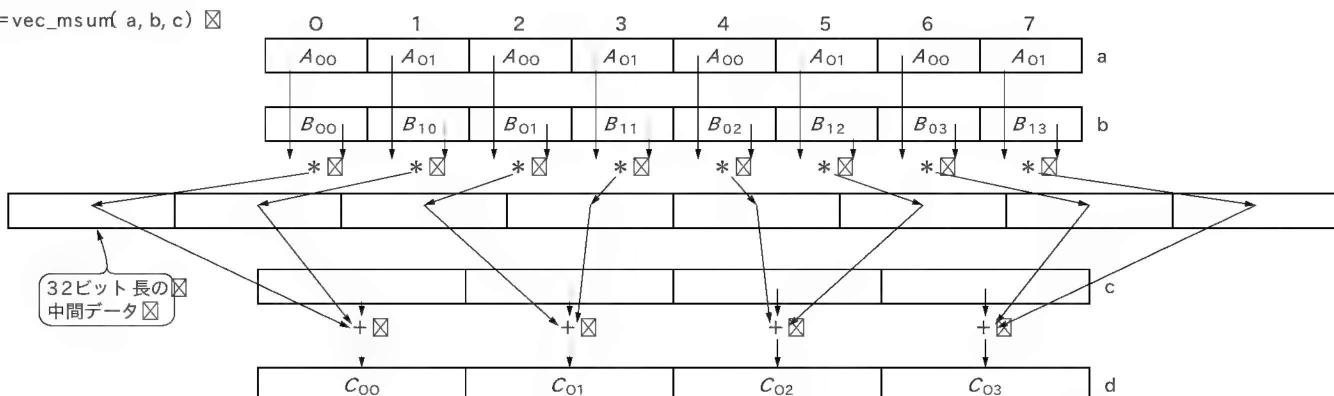
同様の処理をもう一度繰り返すことで，行列Cの2行分の結果がまとめて得られます．この2行分を算出するプログラムをリスト4(p.137)に示します．

完成した行列乗算プログラムを，評価ボードを用いて計測した結果を表3に示します．インライン化による最適化を行った場合で，64回の積和演算が必要となる1回の行列乗算が，データ転送を含めてわずか19サイクルで実行されることがわかります．

〔図10〕vec_msum命令(積和演算)の動作



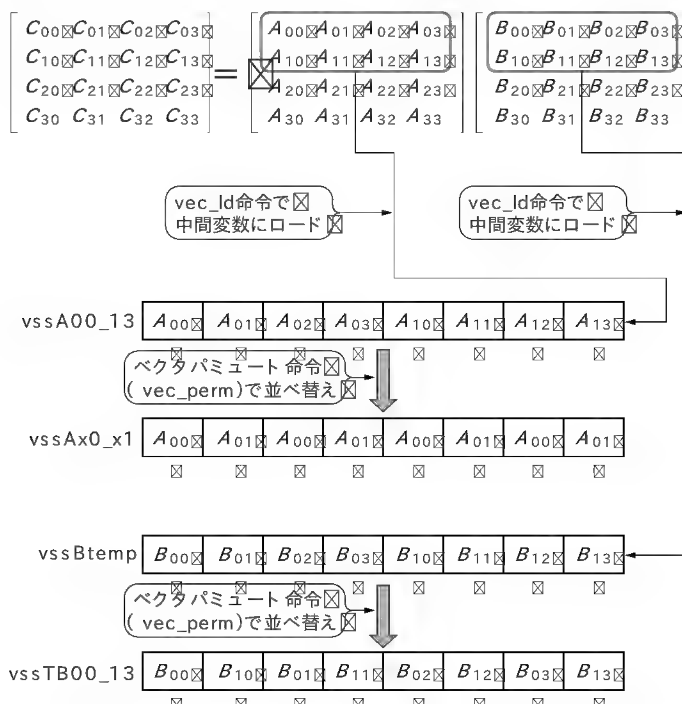

〔図12〕vec_msum命令の適用




〔図11〕行列乗算の並列処理

$$
\begin{aligned}
C_{00} &= A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}\\
C_{01} &= A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31}\\
C_{02} &= A_{00}B_{02} + A_{01}B_{12} + A_{02}B_{22} + A_{03}B_{32}\\
C_{03} &= A_{00}B_{03} + A_{01}B_{13} + A_{02}B_{23} + A_{03}B_{33}
\end{aligned}
$$

〔図13〕ベクタパミュート命令によるデータの並べ替え




〔表3〕行列乗算にかかるCPUサイクル数(16ビット整数形式)

| | インライン化 | 一回の行列乗算あたりにかかるCPUサイクル数 |
|---|---|---|
| 16ビット整数形式の行列乗算結果 | なし | 41 |
| | 実行 | 19 |

## 5. パフォーマンス比較

● 計測環境について

パフォーマンスを計測する環境を構築する際に注意した点は，次のとおりです．その結果，コアの処理性能について，データ転送時におけるL1キャッシュにヒットしない場合のオーバヘッドを除いた正確な数値を算出することが可能になりました．

(1) 処理対象のデータは，データストリームタッチ命令(vec_dst)を使用してあらかじめL1キャッシュに転送

(2) L1キャッシュへの書き込みにともなうキャッシュミスを避

〔リスト3〕並べ替えに使うインデックス情報を定義

```
vector unsigned char vucPermA0 = (vector unsigned char)( 0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3 );
vector unsigned char vucPermA1 = (vector unsigned char)( 4,5,6,7,4,5,6,7,4,5,6,7,4,5,6,7 );
vector unsigned char vucPermA2 = (vector unsigned char)( 8,9,10,11,8,9,10,11,8,9,10,11,8,9,10,11 );
vector unsigned char vucPermA3 = (vector unsigned char)( 12,13,14,15,12,13,14,15,12,13,14,15,12,13,14,15 );
```

けるため，dcbz命令を使用してあらかじめストア用のバッファ領域をゼロクリア

(3) 開発ツールとして，CodeWarrior for Host Processors(version 6.6)を使用し，コンパイルオプションにもっともサイクル数が短縮される Level 4を使用

性能計測は，CPUコアに実装されているレジスタであるタイムベースか，パフォーマンスモニタを使用することになります．タイムベースは，バスクロックに同期して，4バスサイクルごとに1ずつインクリメントされていくカウンタです．パフォーマンスモニタは，CPUサイクル数やキャッシュミスした回数を計測することが可能です．ただし，パフォーマンスモニタは，コアの状態がスーパバイザモードにある場合に使用できます．しかし，MacOS上のアプリケーションは，ユーザーモードで動作するため，PowerMacを使ったサイクルカウント計測には，ユーザーモードでも使用できるタイムベースを使用します．

計測については，計測区間の開始タイミングと終了タイミングでカウンタ値を読み出して差分を計算することで，処理時間を計測することができます．このカウンタ値の読み出し方法は開発ツールによって多少異なります．今回は，PowerMacでも評価が可能なProject Builder用(gcc用)とCodeWarrior用のサンプルコードを用意したので参考にしてください．

PowerMacを使うことで，開発期間を短縮することができますが，性能計測についてはOSの介在によるオーバヘッドが含まれるため，計測結果にばらつきが生じます．そこで，AltiVecで最適化されたサブルーチンの開発および機能評価にはPowerMacを使用し，厳密な性能評価にはMPC7455が搭載されている評価ボードとWindows上で動作するCodeWarrior for Host Processorsを組み合わせたクロスコンパイル環境で行いました．

行列の乗算を行うサブルーチンについては，PowerMacの環境と評価ボードの環境で同一のファイルを使用します．MPC7455とMPC7457とのおもな違いは，最大動作周波数，消費電力，搭載されているL2キャッシュのサイズなので，今回の計測条件では，MPC7457，MPC7447を使った計測でも同様の結果が得られます．

計測は，16個の要素をもつ4行4列の行列同士の乗算を1回とし，80回行った場合と100回行った場合について行いました．それぞれの場合について，1回目のインストラクションキャッシュミスやサイクルカウント計測にともなうオーバヘッドが同じCPUサイクル数分だけ含まれます．そこで，命令と処理データが，すべてキャッシュに存在する場合の厳密なコア処理性能を計測するために，80回行った場合と100回行った場合の差分を用いて，1回分の行列乗算にかかるCPUサイクル数を算出しました．ここで，行列は4行4列であるため，1回の乗算当たりに64回の積和演算が行われることになります．

〔リスト4〕行列2行分の処理(16ビット整数形式)

```
vssA00_13 = vec_ld(0, in1);
in1 += 8; //入力ポインタの更新
vssAx0_x1 = vec_perm( vssA00_13, (vector signed short)(0), vucPermA0 );
vssCx0_x3 = vec_msum( vssAx0_x1, vssTB00_13, (vector signed long)(0) );
vssAx2_x3 = vec_perm( vssA00_13, (vector signed short)(0), vucPermA1 );
vssCx0_x3 = vec_msum( vssAx2_x3, vssTB20_33, vssCx0_x3 );
vec_st( vssCx0_x3, 0, out);
out += 4; //出力ポインタの更新

vssAx0_x1 = vec_perm( vssA00_13, (vector signed short)(0), vucPermA2 );
vssCx0_x3 = vec_msum( vssAx0_x1, vssTB00_13, (vector signed long)(0) );
vssAx2_x3 = vec_perm( vssA00_13, (vector signed short)(0), vucPermA3 );
vssCx0_x3 = vec_msum( vssAx2_x3, vssTB20_33, vssCx0_x3 );
vec_st( vssCx0_x3, 0, out);
out += 4; //出力ポインタの更新
```

〔表4〕行列乗算にかかるCPUサイクル数(浮動小数点形式)

| | インライン化 | ①80回 | ②100回 | 一回の行列乗算にかかるサイクル数 |
|---|---|---|---|---|
| スカラ処理 | OFF | 35,510 | 44,270 | 438 |
| AltiVec | OFF | 3,902 | 4,762 | 43 |
| | ON | 4,095 | 4,695 | 30 |

〔表5〕行列乗算にかかるCPUサイクル数(16ビット整数形式)

| | インライン化 | ①80回 | ②100回 | 一回の行列乗算にかかるサイクル数 |
|---|---|---|---|---|
| スカラ処理 | OFF | 22,910 | 28,513 | 280 |
| AltiVec | OFF | 3,726 | 4,546 | 41 |
| | ON | 3,220 | 3,600 | 19 |

● 性能評価

AltiVecの使用による性能向上を確認するために，最適化が行われていないスカラ処理による計測もあわせて行い，結果を表4と表5にまとめました．

表4から，G4プロセッサによる4行4列の行列乗算処理は，AltiVecで最適化することで，最適化を意識していないスカラ処理に比べて438/30=14.6倍の効率化が確認できたといえます．ベクトル演算ユニットを搭載したG4プロセッサであれば，これまでRISCプロセッサが行うには負荷が大きすぎると考えられていたアプリケーションのディジタル信号処理もデバイス内部で完結させることが可能になります．

ながの・かずひろ　モトローラ(株)半導体事業部 ホストプロセッサ製品部

# 「VxWORKS」を使ったRTOS技術の基礎と応用



第4回

## VxWORKS TCP/IPプロトコルスタックの設計と実装(前編)

濱口 遵一郎

私はVxWORKSユーザー歴の長いソフトウェアエンジニアで，現在はシリコンバレーで仕事をしています．今回はVxWORKS TCP/IPプロトコルスタックの設計と実装について，VxWORKSユーザーの視点から解説します．VxWORKSネットワークのデザイン背景など，極力マニュアルに記述されていないことを中心に話を進めていきます．実際にVxWORKSネットワークを使っている方にも読み応えのある内容になるように意識して書きました．

まず前編は組み込みシステム開発プラットホームとしてのTornado/VxWORKSの特徴と，BSDとVxWORKS TCP/IPプロトコルスタックの比較について解説します．後編はVxWORKS TCP/IPプロトコルスタックの設計と実装について詳しく説明していきます．

## VxWORKSネットワーキングについて

前回の記事で指摘があったように，VxWORKSにBSD TCP/IPプロトコルスタックを移植したのは先見の明があったと思います．実際，移植した当時は，"ネットワーキング＝BSD"という図式が成り立つほど，BSDが与えた影響力は大きかったと思います．WindRiverを筆頭にほとんどのRTOSベンダがBSD TCP/IPプロトコルスタックの移植を行ったのはこういった歴史的な背景があると筆者は考えます．

### ● 組み込みネットワーク機器に特化したOS

ではVxWORKSのTCP/IPプロトコルスタック実装がほかのBSD系のものより秀でている点は何でしょうか．筆者が考えるに，それはユニークさではないでしょうか．オープンソースコミュニティには見当たらない，組み込みネットワーク機器に特化した事例の中からいくつか紹介します．

- Network Protocol Tool Kit(以下NPT)に代表されるようにユーザーが自由にプロトコルやネットワークをプラグインできる柔軟な設計思想
- あるワイヤレスDSLルータに使われているようなWindNet PPP，PPPoEやRadius Clientなど完成度の高いネットワークコンポーネント群
- ソフトウェアベースのfast-path forwarding，Unnumbered I/FやRFC1812として定義された重要な機能群を搭載した，レイヤ3パケットフォワーディング向けのルータスタック
- Tornado Home GatewayやTornado Platformに代表されるような，Time-To-Marketを強く意識したプラットホーム

これらを見ると，他社に先駆けて新しい分野の製品をリリースして，競合他社もそれを追う形になっています．つまりWind Riverをウォッチしていると組み込みソフトウェア開発環境のトレンドが把握できます．

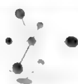
ではネットワークに限定して考えた場合，VxWORKSはどのようなネットワークデバイスや組み込みソフトウェアに向いているのでしょうか．割り込み応答が保証されているHard RTOSとTCP/IPプロトコルスタックとの関係という観点で見ていきましょう．

### ● Hard RTOSとロードモジュールサイズ

RTOSベースのネットワークの事例として，リアルタイム性が要求されるController Area Network(CAN)[注1]などが考えられます．また，VoIPのコアとなるRTP/RTCPは非RTOSでも実現可能です．電話の通話をインターネット経由で行うため，プロトコルとしては厳密なリアルタイム性を要求されませんが，IP電話端末についていえば通話品質を確保するにはHard RTOSのほうが設計がしやすいと思います．要求仕様を満たすための最低限のハードウェア仕様も見積もりやすいのでシステムのコストダウンが容易になります(ロイヤリティの話は別として)．CANやRTP/RTCPに限らず，精密な仕様が要求されるネットワーク製品には筆者はHard RTOSをお奨めします．さらにハードウェアオフローディングを追求するのであれば，ネットワークプロセッサの採用も検討すべきでしょう．

また，大きくなったと言われるVxWORKSですが，いまだ

注1：Controller Area Network(CAN)は1989年にRobert Bosch GmbH社により開発され，ISOで国際的に標準化されたシリアル通信プロトコルである．元来，自動車業界での使用のために開発されたCANシリアルバスシステムは，ますますビルオートメーション，医療機器，海洋関係の電子機器での利用が増加している．



に Linux/BSD に比べてロードモジュールサイズが小さいので小型のネットワークデバイス，たとえばワイヤレス MP3プレイヤのプラットホームとしても適していると思います．とくにスケーラビリティと SMP コンピュータ上でのパラレル度の向上を競い合っている Linux と FreeBSD がモジュールサイズの縮小化に方向転換するとは考えられず，これらのオープンソースソフトウェアをベースとして組み込みシステム向けのソリューションを提供しているソフトウェアベンダはサイズ縮小化のためにさらなる努力が必要になります．

● リアルタイム性はどこまで保証できるか

よくある質問ですが，割り込み応答が保証されている RTOS の TCP/IP プロトコルスタックならパケットの取りこぼしがなくなるか否か——精密に言うと，答えは NO です．RTOS の ISR ですべて処理できるほど TCP/IP は短小なプロトコルではありません．ISR での処理を極力短くしてもタスクレベルでの処理が追いつかなければ，Ethernet コントローラからの割り込みをすべて処理できていても，パケットの取りこぼしは発生します．

では Linux や BSD のような非 RTOS ではどうでしょう．割り込み禁止の期間が長くかつ算定不能なため，当然割り込みレベルでのパケットの取りこぼしが発生する可能性があります．しかし，割り込みを受け付けるよりは現在カーネル内で処理しているジョブを完了することに専念するというシンプルなスケジューリングアルゴリズムのため，リッチなハードウェアを与えることにより性能向上が期待できます．

Ethernet はフレーム消失を容認していますし，TCP/IP 自体もパケット消失時の再送を試みますが，コミュニケーションの絶対的信頼性を保証しているわけではありません．しかし，パケット消失がどこで発生しうるかの考察は興味深いと思います．

## zero-copy TCP か否か

最近では zero-copy TCP は珍しくありませんが，VxWORKS TCP/IP プロトコルスタックには昔から BSD ソケット API と ZBUF と呼ばれる zero-copy TCP のソケット API があります．VxWORKS ネットワーキングの特徴の一つである“Binding Concept”はここでも活かされており，ユーザーは BSD ソケット API と ZBUF ソケット API をバックエンド[注2]としてソケットレイヤにバインドできます．このデザインの優位点については別章で論じます．

ただし，筆者はパフォーマンス追求に関してはバランス感覚が必要だと考えています．VxWORKS の ZBUF，または巷で売られているプロトコルスタックの zero-copy TCP API を使うということはアプリケーションがその API に縛られてしまうということを意味します．zero-copy TCP API を使うなとは言いませんが，zero-copy TCP API によって得られる性能差とそのトレードオフ[注3]を注意深く吟味してください．

VxWORKS TCP/IP プロトコルスタック本体では copy は行っていません．BSD ソケット API を使った場合の受信パスでは tNetTask がドライバのリングバッファからパケットを取り出し，mbuf[注4]（VxWORKS では mBlk と呼ぶ）に形成してからは TCP/IP プロトコルスタック内ではポインタによるデータの受け渡しを行っているため，データコピーは発生しません．

自分で zero-copy TCP を実装したいが，ネットワークプロトコルスタックを一から書くのはどうも，というエンジニアの方には Network Protocol Tool Kit（以下 NPT）でソケットレイヤの部分だけ実装して VxWORKS のネットワークプロトコルスタックとインテグレーションすることをお勧めします．詳しい方法については関数 `sockLibAdd()` の使用方法，sockLib とソケットバックエンドとのインターフェースを含めて後編で解説します．

## ネットワークをフルに活用した Tornado 開発環境

VxWORKS ユーザーは早くからネットワークを活用した開発環境を利用してきました．筆者は VxWORKS を使う前は M68K CPU を用いたターゲット上で HP64000 シリーズの ICE を活用していましたが，ターゲット CPU が RISC になったのをきっかけに VxWORKS を使うようになりました．今でこそ，JTAG/BDM デバッガが普及していますが，その当時は VxWORKS のターゲットシェル，ローダそしてネットワークデバッグエージェントにとても感激したのを覚えています．シェルスクリプトが使えたり，netDrv や NFS によって開発ターゲットハードウェアからホスト EWS のファイルシステムに透過的にアクセスできたり，telnet や rlogin ができたりと，まるで UNIX 上でアプリケーションを開発するような便利さでした．

WindView のようなイベントトレースツールや VxSim のようなシミュレータが提供され，さらに関数コールがシェル上で可能だったことなど，当時の UNIX 実装でもなかったので，ある意味 UNIX をも超えていたといえるでしょう．そういったわけで VxWORKS のファンになるまでにそれほど時間がかかりませんでした．

注2：ソフトウェアが2階層以上で構成されているケースで，ユーザーもしくはユーザーアプリケーションとのインターフェースをフロントエンドといい，対してフロントエンドよりも後方に位置するものをバックエンドと称する．たとえば，低水準入出力関数 Read/Write がフロントエンドで，その下で動くハードウェアごとの実際の I/O 処理がバックエンドともいえる．

注3：現在 Proposed Standard である RFC 3390 Increasing TCP's Initial Window のように，TCP/IP プロトコル自体にもまだまだパフォーマンス改善の余地があり，IETF を中心に精力的な活動が行われている．

注4：mbuf とは BSD の IPC 用メモリ管理モジュールとその構造体を指す．BSD におけるネットワークパケットは mbuf によってハンドリングされる．

netDrvとはFTPとRSHをバックエンドとするVxWORKSオリジナルの仮想ネットワークドライブです．netDrv自身はネットワーク絡みのコードを一切抱えておらず，シンプルなWrapper Moduleです．初期化ルーチン `netDrv()` を呼ぶと `ioDrvInstall()` で登録された各netDrv I/OルーチンはVxWORKSブート時に選択されたブート方法によってFTPかRSHのルーチンを呼び出します．たとえば，netDrvに対して `ls` を実行すると，FTPモードのときには `ftpXfer()` で `NLST` をホストに送信し，RSHモードのときには `rcmd()` で `/bin/ls -a` を送信します（だから `/bin` に `ls` がないとエラーとなる）．いずれにしても，FTPでは `ftpXfer()` を，RSHでは `rcmd` でホストと通信しています．NFSのようにルートの権限がなくても，PCでも手軽にリモートファイルシステムを使用できます．もちろんパフォーマンスや効率を考えた場合はNFSを使用すべきですが，こういったところに開発エンジニアのスマートさ，斬新さが伺えますし，自分が実際にプログラムを書くときの参考になります．

## WDB/WTX —— Tornadoのデバッグバックエンド

さて，シリアルバックエンド経由でデバッグを行っていたのが当たり前の時代に，ネットワークバックエンドでホストとターゲットを接続したり，シェルやシンボルテーブルをホスト側にもたせて，最小メモリしか実装していないターゲット上でのデバッグを可能にしたりと，VxWORKSからTornadoへの開発環境の進化は後にほかのRTOSにも大きく影響を与えました．その開発環境の進化を支える技術の一つであるWind Debug Protocol Agent（WDB Agent）の中で，とくにENDモードのWDB Agent（WDB_COMM_END）について解説します．

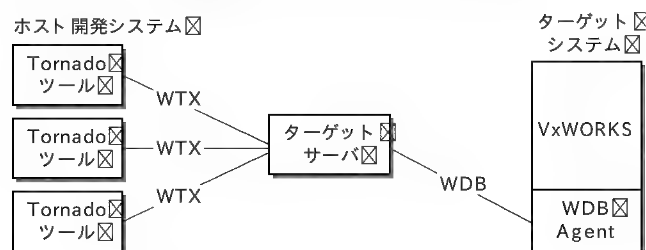
● WDBデバッグエージェントの動作

図1，図2にWDB Agent，Target Server，そしてWindRiver Tool Exchange Protocol（WTX）を介してターゲットと通信するTornado Tool群の関係を概念図で示します．WDB Agentはひと言でいうとミニUDPスタックで，ターゲットサーバとの通信に使われるWDBプロトコルはXDR/RPCがベースとなっています．WDBプロトコルで特筆すべきなのが，コンパクトなGopherの実装によりターゲットのデバッグ情報（メモリの内容やブレークポイントの設定など）を取得･操作するために，そのためのモジュールをターゲット側に用意する必要がないことです．またXDR/RPC上での実装を行った結果，ユーザーはホスト－ターゲット間のCPUやハードウェアの違いを意識する必要がなくなりました．なお，WDBデバッグエージェントはtWdbTaskというタスクプライオリティ3という非常に高い優先順位で動作するタスクとして実装されています．

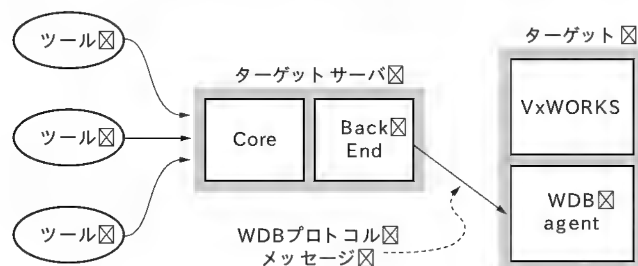
読者の方は「VxWORKSにはtNetTaskというTCP/IPプロトコルスタックがあるのに，なぜデバッグエージェントとして別途TCP/IPプロトコルスタックがいるのか」と疑問に思うかもしれません．逆に言うと，デバッグエージェントの通信をtNetTaskが受けもつとどんな不都合があるのでしょう．たとえば，自分でプログラムしたソケットアプリケーションのデバッグを考えてみましょう．適当な場所にブレークポイントを掛けてそのプログラムを走らせてみます．ブレークポイントにヒットしたらホストデバッガとデバッグエージェントの通信が切断されませんでしたか．プロトコルスタックに依存したデバッグエージェントはネットワークアプリケーションのデバッグには使えないのです．これで独立したミニUDPスタックとして実装されたWDBエージェントの必然性，とくにENDモードとシリアルモードのWDBの必然性を理解していただけたと思います．

では図3を参照しながらWDB ENDモード（WDB_COMM_END）を解説します．WDB_COMM_ENDの構造を知ることで後述するMUX/ENDについて理解が深まります．ENDモードのWDBエージェントはSNARFプロトコルとして実装されており，t NetTaskより先にパケットを取得してそれがWDB

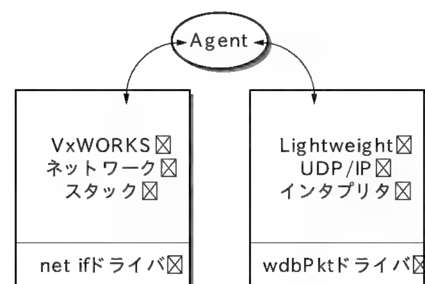
〔図1〕Tornadoを構成するプロトコル —— WTXとWDB

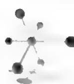



〔図2〕Target SeverとTargetの関係 WDB Back End




〔図3〕WDB UDPスタックとtNetTaskの関係





パケットであればWDBのミニUDPスタック内で処理し，そうでなければ，通常パケットとみなし，t NetTaskに渡します．ENDドライバは割り込みモードとPollモードに動的に切り替わることができ，Pollモードで動作させることにより高速なネットワーク上で強力なTornadoのホストツール群を用いてシステムモードデバッギングが可能です．大半のデバッグはENDモードで快適に行えますが，ネットワークインターフェースドライバや一部のMUX/ENDコード（t NetTaskとWDBでコードをシェアしている部分）のデバッグにはシリアルモードのWDBを使用します．

### ● ハードウェアに依存したパッケージ Board Support Package

図4はVxWORKSのモジュール構成を示した概観図です．VxWORKSにおいて，ハードウェアに依存したモジュールがBoard Support Package（以下BSP）を中心に構成され，ハードウェア非依存部分とは分離されているのがわかります．Ethernetなどのネットワークデバイスのハードウェア初期化はEnhanced Network Driver（以下END）とは別のモジュールで行われ，そのソースファイルはBSPの一部としてVxWORKSビルド時に`#include`文によって`sysLib.c`から取り込まれます．

BSPというのはUNIXやWindowsの世界ではあまりなじみのない概念ですが，組み込みソフトウェアの世界ではBSPはOS選定の重要なファクタとなっています．BSPはターゲットとなるハードウェアを直接制御するモジュールの集合体です．VxWORKSのBSPは次のファイルから構成されます．ここではEthernetコントローラの初期化に重点を置いて解説し，他の部分は割愛します．

- `romInit.s`…VxWORKSのブートROMスタートアップコード
- `sysALib.s`…ハードウェアに最適化されたアセンブラルーチン群
- `sysLib.c`…ハードウェア制御を行う標準化されたAPI群
- `sysNet.c`…GENERICなNICハードウェア初期化ルーチン群
- `sysXXXEnd.c`…NIC固有のハードウェア初期化ルーチン群

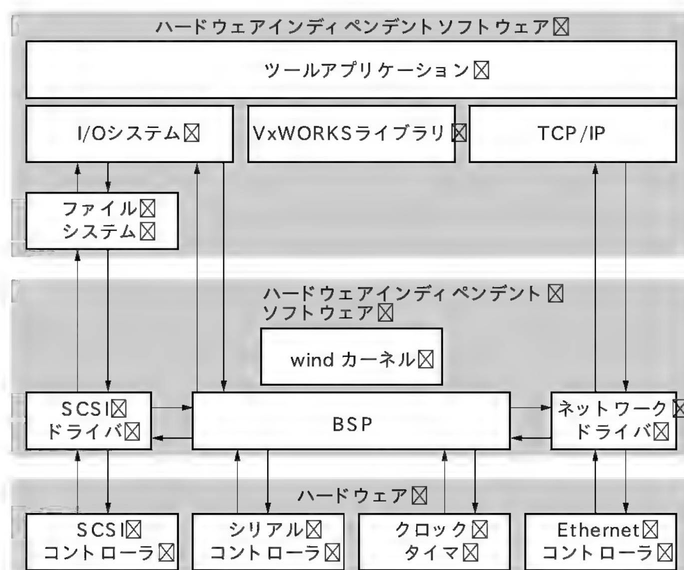
`sysNet.c`は`sysLib.c`から`#include`によってBSPの一部としてコンパイル時に取り込まれます．複数種類のNICをサポートするBSP（例：pentium）では`sysNet.c`からsysで始まるEthernetコントローラハードウェア初期化モジュール（例：`sysFei82557End.c`，`sysEl3c90xEnd.c`）を複数includeします．

実際のルーチンのコードはハードウェアに依存しますが，`sysNet.c`では基本的に以下の処理を行います．

- PCIレベルでのEthernetコントローラハードウェアの初期化
- EthernetコントローラのメモリとI/OをCPUのアドレススペースにマッピング
- NIC固有のハードウェア属性データの初期化

VxWORKSはUNIXではないので，ルートファイルシステムも必要ありませんし，個々のモジュールも組み込みソフトウェアの命題であるフットプリントの最小限化を強く意識した作りになっているので，スタティックリンクを行ってみたら，ランタイムロードモジュールが驚くほど大きくなっていた，などということはありません．

〔図4〕VxWORKSのモジュール構造




## なぜBSDが良いのか ── BSDソケットAPI，The Internet DeFacto Standard

BSDのネットワークの利点はインターネットの発展の歴史で技術的に裏打ちされているところです．The Design and Implementation of the 4.4 BSD Operating System[4]（4.4BSDの設計と実装）で紹介されていますが，インターネットは，一つの端末と一つの端末で通信が最適に行われるだけでは成立しません．無数の端末，無数のルータ，また経路が確立されておらず，通信のバンド幅，経路数，品質が不明なインターネットでは，輻輳（ふくそう）状態を避け全体のパフォーマンスの最適化を達成することも目標の一つです．輻輳状態とは，ネットワークの1経路が混雑しパケットロスにより再送が多発し，その再送によりますますパケットロスが発生し，ネットワークがダウンしてしまうことです．

そのためには，パケットロスした場合は，端末はネットワーク負荷に耐えられないのだと認識してパケットの送出を少しずつ抑えないといけません．車社会でいえば，譲り合いの精神でしょうか．今でもインターネットを支えるのはBSDの流れを組むOSです．これがBSD系ネットワークのプロトコルスタックの利点といえます．

あえて説明が必要だとは思いませんが，The Internetという

言葉が世に知られる前からBSDはネットワーキングのデファクトスタンダードでした．BSDプロトコルスタックは後に世に出たTCP/IPプロトコルスタックに多大な影響を与え，その大半はBSDソケットとの互換性をうたっていますし，ネットワークアプリケーションの大半はBSDソケットAPIを使って書かれています．BSDソケットAPIとの親和性を活かしたオープンソースベースのアプリケーション移植とインテグレーションの容易さは特筆すべきでしょう．時間がある人はBSDソケットアプリケーションをWindows CEとVxWORKSに移植してみて，どちらが楽か比較してみると良いでしょう．

これに関連して，パフォーマンス至上主義でBSDとソケットレベルの互換性を軽視あるいは無視しているプロトコルスタックとソケットレイヤの実装を使うことはソケットAPIを使う恩恵をみずから破棄していることになります．絶対禁止とは言いませんが，熟慮のうえ，最良の選択肢を選んでください．

また，BSDはIP（Intellectual Property）保護がGPLベースの場合に比較して容易なことなどです．すでに組み込み市場でLinuxベースのデバイスが出回り始めているので，あまりGPLに関して過剰に神経質になる必要はないと思いますが，注意事項であることは確かです．

## VxWORKSと4.4BSDの違い

VxWORKSネットワークはBSDとどう違うのでしょうか．

〔表1〕VxWORKSと4.4BSDの違い

| 機　能 | VxWORKSの実装 | BSDの実装 |
|---|---|---|
| ROM化 | 可能 | 可能 |
| フットプリント | VxWORKS+TCP/IPで360Kバイト以下．もっと小さくなる | FD1枚に収まるが，そこから小さくするのは難しい |
| BSPサポート | 豊富 | 発展中 |
| Networkサービス | タスク（Thread） | Background Process |
| Networkイベント処理 | Deterministic | Best Effort |
| ネットワークバッファ管理 | 固定（サイズ，速度重視） | 可変（拡張性重視） |
| ネットワークバッファ管理ライブラリ内での排他制御 | intLock（割り込み禁止，速度優先とPollモード対応のため） | vm_mapに対するロッキング．Paging Delayあり |
| ネットワークバッファのWAIT方法 | セマフォ（NOWAITかWAIT FOREVERのどちらか） | Sleep/Wakeup関数（柔軟性重視） |
| sobufロッキング | セマフォ（速度重視） | Sleep/wakeup関数（柔軟性重視） |
| Set Priority Level | セマフォ（速度重視） | 割り込みレベル設定 |
| 受信ISRでの処理 | ISRはJobキューイングのみ | ISRである程度処理する |
| ドライバモデル | END/NPT（柔軟，拡張性） | Classic（現状問題なし） |
| プロトコルバインディング | MUX（柔軟） | Classic（必要性なし） |
| Build Configuration | Protocol単位（柔軟） | 可能だが，制限あり |

この違いを考察することによって組み込みネットワークの要件が自ずと導き出されます．以下にVxWORKSネットワークとBSDの違いを示します．

表1に示すとおり，VxWORKSのTCP/IPプロトコルスタックがBSDの実装とあちこち違います．これはVxWORKSが組み込みシステム用途を強く意識した作りになっているのに対し，ハイパフォーマンスコンピューティングの分野で活躍しているFreeBSDを初めとするBSD実装とはターゲットとしている分野の違いがデザインにも反映されています．

それでは組み込みネットワークの要件とは何でしょうか．筆者が考えるに次の点は外せないと思います．

- ROM化が可能であること
- フットプリント（リソース使用量）が要求仕様を満たすこと

たった二つだけと首をかしげる方もいるかと思いますが，究極的に言えば，ROM化が可能で，実装予定のRAM/ROMにシステムが収まれば，とりあえずはOKと言えます．CANのようなリアルタイム性をシビアに要求するネットワークは別として，一般的な組み込みネットワーク機器にはネットワークイベント処理（NICからの割り込みなど）がDeterministicである必要はありませんし．同様にすべての組み込みネットワーク機器に対してパフォーマンスがシビアに要求されているわけでもありません．

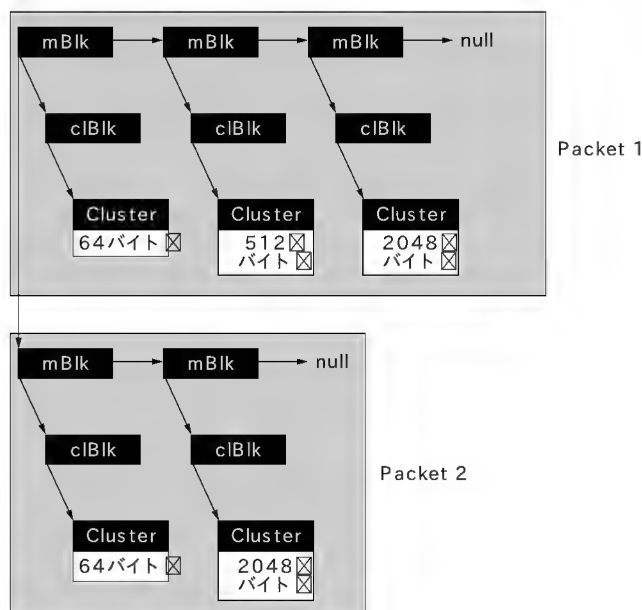
ではROM化できてフットプリントが要求仕様を満たしたら，ほかにどのような要素がOSやTCP/IPプロトコルスタックを選定する決め手となるのでしょうか．ここでの解説はVxWORKSとBSDの間で択一的な選択をする目的ではなく，あくまでVxWORKSをBSDと比較することでBSDをベースとするVxWORKSのプロトコルスタックが組み込み向けにどのように進化してきたかを考察することを目的とします．

VxWORKSのタスクモデルは他のオペレーティングシステムアーキテクチャ上でいうところのスレッドに該当します．タスクコンテキストスイッチや割り込み遅延が予測可能なため，ネットワークイベントに対する応答性がシビアになればなるほど，RTOSの代表格であるVxWORKSを使う意味がでてきます．

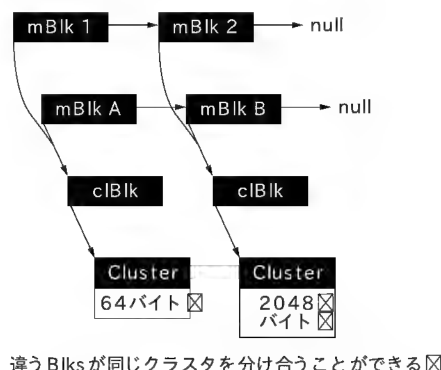
ネットワークバッファ管理に関してもBSDとVxWORKSは違います．図5～図7を見るとmbufとの違いがよくわかります．BSDのmbufはデータエリアを包含していますが，VxWORKSのnetBufLibは制御構造体mBlk，ClBlkとデータ領域であるクラスタを分割して管理しています．TCP/IPではtinygram[注5]以外はMSS目一杯のサイズでデータを送受信するので，mbufで内含されているデータ領域では収まりきらず，結果としてmbufチェインを駆使することになります．mbufチェインが発生するのはVxWORKSでも同じですが，BSDがmbufとクラスタ（M_EXT）の2データ領域しかもたないのに対し，VxWORKSではユーザーが複数のクラスタサイズを設定できるのは便利だと思います．

VxWORKSのTCP/IPプロトコルスタック内でのマルチス

〔図5〕VxWORKS netBufLibネットワークバッファ概念図

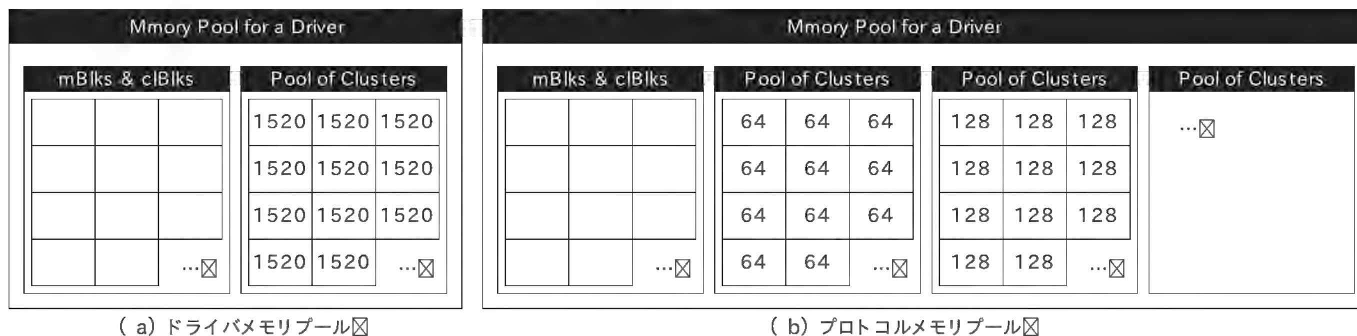



〔図6〕mBlksのバッファチェイン概念図




違うBlksが同じクラスタを分け合うことができる

〔図7〕netBufLibによるドライバとプロトコルのバッファ分割管理概念図




(a)ドライバメモリプール　　(b)プロトコルメモリプール

レッドに対するデータ保護はセマフォというシンプルかつ非常に高速なロッキングメカニズムによって行われています．対してBSDでは`sleep/wakeup`関数を使って，システムの柔軟性を重視しています．

たとえば，ソケットのブロッキングI/OはBSDだと`sleep`関数によって実現されていますが，一方のVxWORKSでは高速かつシンプルなセマフォによって実現されています．

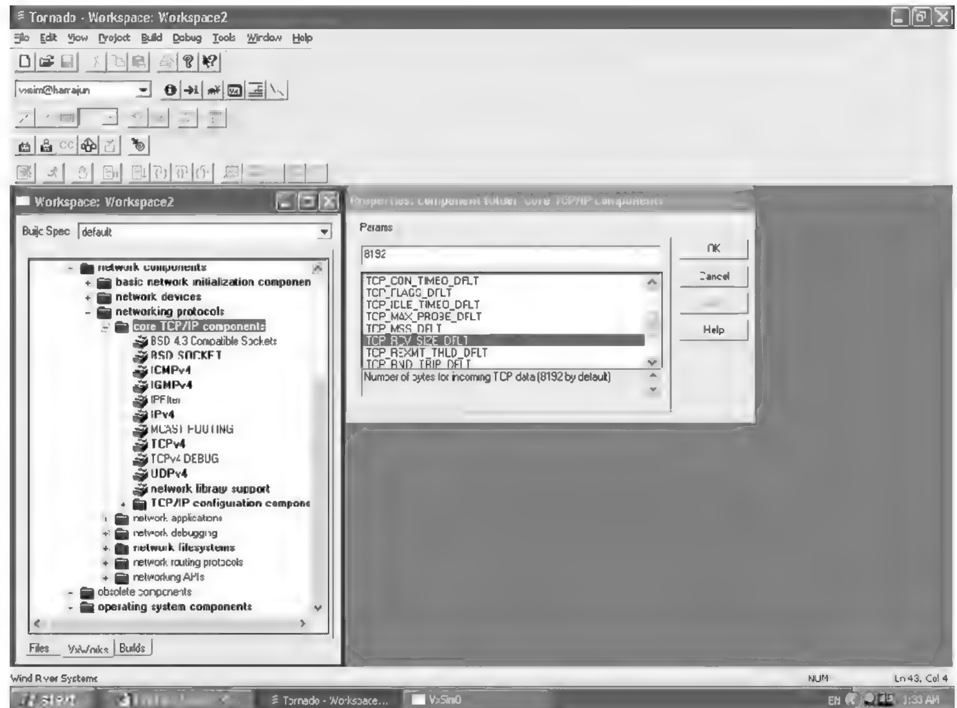
ネットワーク割り込みが発生したときも，VxWORKSはイベントをTCP/IPプロトコルスタックのタスクのジョブキューにキューイングして直ちに終了します．それに対して，BSDはある程度のパケット処理をネットワークISRの中で行います．ここでもRTOSとTSSであるUNIXライクなBSDとの違いが対照的です．同様に，SPL(Set Priority Level)の実装もBSDとVxWORKSでは異なります．BSDでは割り込み禁止レベルの変更処理ですが，VxWORKSではSPLはセマフォです．スケジューリングが割り込みベースのVxWORKSではネットワークバッファ取得(これもPollモード対応のため)など非常に限定されたケースを除き，割り込みを禁止しません．

WindRiverのMUXアーキテクチャは後述しますが，BSDの`if_attach()`コールによるバインディングメカニズムをさらに発展させたものです．たとえば，これにより，デバッグエージェントをSNARFプロトコルとして登録し，ポールモードにてパケット受信を行うことでネットワーク経由でのシステムモードでバッギングを実現しています．

Network Protocol Tool Kit(以下NPT)はMUX/ENDをさらに発展させたもので，リンク層に依存する処理をすべて独立し

注5：tinygramとはその名のとおり小さいパケットを指す．よく引き合いに出されるのがtelnetのようなインタラクティブなTCPソケットアプリケーションで送受信されるデータサイズが1バイトのパケットだが，RFC896(別名Nagle Algorithm)の見地からいうとMSS(Maximam Segment Size)以下のTCPセグメントであれば，tinygramと考えるべきだろう．逆にMSSのTCPパケットを通常Full Segmentと呼ぶ．

〔図8〕
Tornado Project Facility によるネットワークコンフィグレーション




たコンポーネントとし，バインディングメカニズムはリンク層非依存とすることで，Ethernet と TCP/IP，BSD ソケット以外のネットワークでも VxWORKS の既存のネットワークソフトウェア資源を最大限活用することを可能にしています．

VxWORKS はプロトコル単位で取り外しができます．たとえば，TCP 抜きの TCP/IP プロトコルスタックも作れます．BSD はインターネットプロトコル群用の PROTOSW にプロトコルが登録されているので，取り外しできません．ちょっとしたことですが，いらないモジュールは取り込まない，という組み込みネットワークならではの配慮です．

TCP/IP プロトコルスタックやネットワークアプリケーションの設定の大半は Project Facility で行えます．GUI で設定をするのが好きなユーザーは Tornado から Project Facility を操作し，従来どおりの方法で BSP ごとに `config.h` 設定ファイルを編集して VxWORKS をビルドすることができます(図8参照)．

以上，VxWORKS TCP/IP プロトコルスタックが BSD をベースに組み込み機器向けの TCP/IP プロトコルスタックとして進化してきたことがわかったと思います．

## まとめ

前編として駆け足で Tornado のネットワークをフルに活用した組み込み開発環境から VxWORKS と BSD の TCP/IP プロトコルスタックの違いを説明しました．これをきっかけに VxWORKS に興味をもっていただければ幸いです．後編は VxWORKS TCP/IP プロトコルスタックの設計と実装と題してかなり突っ込んだ話を展開してゆくつもりですので，ご期待ください．

参考文献


●VxWORKS ネットワーキングについて～Board Support Package
1) Tornado User's Guide, WindRiver
2) VxWORKS Programmer's Guide, WindRiver
3) VxWORKS Network Programmer's Guide, WindRiver
●VxWORKS と 4.4 BSD の違い
4) The Design and Implementation of the 4.4BSD Operating Systems, McKuisk, Bostic, Karels, Quarterman, AddisonWelsley
5) TCP/IP Illustrated Vol.1, Stevens, Addison Welsley
6) TCP/IP Illustrated Vol.2, Stevens, Addison Welsley
●自身の理解を再確認するために参考にした文献
7) UNIX Networking Programming, Stevens Prentis-Hall
8) 4.4BSD UNIX Networking Internals, Mike Karels, BSDI,Inc



はまぐち・じゅんいちろう　カリフォルニア州バークレー在住プログラマ
E-mail： inquire2hama@yahoo.co.jp

x86CPUだけでもマスタしたい

# 開発技術者のためのアセンブラ入門

## 第23回　SIMD命令（2）SSE/SSE2命令（その1）　大貫 広幸

前回（2003年12月号）に引き続き，今回は，Pentium IIIで追加されたSSE命令，そしてPentium4で追加されたSSE2命令について，その概要を説明します．

### SSE/SSE2で追加されたSIMD命令

初めでも述べたようにSSE命令はPentium III以降，SSE2命令はPentium4以降で使用可能となったSIMD命令です．実際のプログラムでは，前回コラムで説明したCPUID命令を使い，SSEあるいはSSE2命令が使用可能か否かを調べる必要があります．

SSE命令では，主としてパックド単精度浮動小数点に対するSIMD命令が追加されました．そして，SSEに続くSSE2命令では，パックド倍精度浮動小数点に対するSIMD命令と，128ビット長のパックド整数に対するSIMD命令が追加されました．

表1と表2は，SSEおよびSSE2で追加された命令を一覧にしたものです．表1は浮動小数点を扱う命令，表2は浮動小数点以外の命令を示したものです．

● SSE/SSE2命令で扱うデータ

SSE命令で扱う浮動小数点は，IEEE754で規定されている単

〔表1〕SSE/SSE2の命令一覧（浮動小数点命令）

| 分類 | データの構成 | インストラクション名（ニモニック） | |
|---|---|---|---|
| | | SSE命令 | SSE2命令 |
| データ転送命令 | パックド | MOVAPS, MOVUPS, MOVHPS, MOVLPS, MOVHLPS, MOVLHPS, MOVMSKPS | MOVAPD, MOVUPD, MOVHPD, MOVLPD, MOVMSKPD |
| | スカラ | MOVSS | MOVSD |
| 変換命令 | パックド | CVTPI2PS, CVTPS2PI, CVTTPS2PI | CVTPI2PD, CVTPD2PI, CVTTPD2PI, CVTDQ2PD, CVTPD2DQ, CVTTPD2DQ, CVTPS2PD, CVTPD2PS |
| | スカラ | CVTSI2SS, CVTSS2SI, CVTTSS2SI | CVTSI2SD, CVTSD2SI, CVTTSD2SI, CVTSS2SD, CVTSD2SS |
| パックド単精度浮動小数点命令 | パックド | | CVTDQ2PS, CVTPS2DQ, CVTTPS2DQ |
| シャッフル命令とアンパック命令 | パックド | SHUFPS, UNPCKHPS, UNPCKLPS | SHUFPD, UNPCKHPD, UNPCKLPD |
| パックド算術命令 | パックド | ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, RSQRTPS, MAXPS, MINPS | ADDPD, SUBPD, MULPD, DIVPD, SQRTPD, MAXPD, MINPD |
| | スカラ | ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, RSQRTSS, MAXSS, MINSS | ADDSD, SUBSD, MULSD, DIVSD, SQRTSD, MAXSD, MINSD |
| 比較命令 | パックド | CMPPS | CMPPD |
| | スカラ | CMPSS, COMISS, UCOMISS | CMPSD, COMISD, UCOMISD |
| 論理演算命令 | パックド | ANDPS, ANDNPS, ORPS, XORPS | ANDPD, ANDNPD, ORPD, XORPD |

〔表2〕SSE/SSE2の命令一覧（浮動小数点以外の命令）

| 分　類 | インストラクション名（ニモニック） | |
|---|---|---|
| | SSE命令 | SSE2命令 |
| 64ビット SIMD整数命令 | PAVGB, PAVGW, PEXTRW, PINSRW, PMOVMSKB, PSHUFW, PSADBW, PMULHUW, PMAXUB, PMAXSW, PMINUB, PMINSW | |
| 128ビット SIMD整数命令 | | 128ビット化されたMMX命令, MOVDQA, MOVDQU, MOVQ2DQ, MOVDQ2Q, PSHUFLW, PSHUFHW, PSHUFD, PUNPCKHQDQ, PUNPCKLQDQ, PADDQ, PSUBQ, PMULUDQ, PSLLDQ, PSRLDQ |
| キャッシュのフラッシュ | | CLFLUSH |
| キャッシュ制御命令 | MASKMOVQ, MOVNTQ, MOVNTPS | MASKMOVDQU, MOVNTDQ, MOVNTPD, MOVNTI |
| プリフェッチ命令 | PREFETCHh | |
| 命令順序付け命令 | SFENCE | LFENCE, MFENCE |
| PAUSE | | PAUSE |
| ステート管理命令 | LDMXCSR, STMXCSR, FXSAVE, FXRSTOR | |

精度の実数のみとなっています．この単精度の実数を四つ連続させた「パックド単精度浮動小数点」で演算を行います．単精度の実数（単精度浮動小数点）は，一つの値で32ビット（4バイト）使用するので，パックド単精度浮動小数点は，128ビット（16バイト）の長さとなります．そのため，SSEのパックド単精度浮動小数点の演算では，128ビット長のXMMと呼ばれるレジスタが使用されます．

SSE2命令で扱う浮動小数点は，同じくIEEE754で規定されている倍精度の実数となります．倍精度の実数（倍精度浮動小数点）は，一つの値で64ビット（8バイト）使用します．SSE2でも，SSEで使われる128ビット長のXMMレジスタを使用するため，SSE2の「パックド倍精度浮動小数点」は，倍精度の実数を二つ連続させたものとなります．

SSEが扱う単精度浮動小数点，SSE2が扱う倍精度浮動小数点の値は，共にx86系CPUがもつFPU（浮動小数点演算ユニット）が扱う浮動小数点と互換性があります．そのため，SSEやSSE2が演算処理した浮動小数点値をFPUで演算処理することができますし，その逆にFPUで演算処理した浮動小数点の値をSSEやSSE2で演算処理することもできます．

SSE2では，パックド倍精度浮動小数点の演算の他に，128ビット長のXMMレジスタを利用した，128ビット長のパックド整数に対する「128ビットSIMD整数命令」も追加されています．この128ビットSIMD整数命令は，前回説明したMMXの64ビット長のパックド整数の命令を，128ビット化したものです．

そのため，128ビットSIMD整数命令では，パックドバイト整数はバイト値が16個連続したデータ，パックドワード整数はワード値が8個連続したデータ，パックドダブルワード整数はダブルワード値が4個連続したデータとなります．また，新たなパックド整数として，SSE2では2個のクワッドワード値が連続した「パックドクワッドワード整数」と128ビットを一つの値とする「ダブルクワッドワード整数」が追加されました．

図1は，SSEおよびSSE2で使用されるデータを図で表したものです．

● パックドとスカラ

SSE/SSE2命令では，パックされた浮動小数点（パックド浮動小数点）の他に，「スカラ」と呼ばれる浮動小数点（スカラ浮動小数点）も扱います．そのため，表1のSSE/SSE2の浮動小数点命令の一覧では，命令をパックドとスカラに分けて表しています．

スカラは，パックされていない，独立して存在する一つの浮動小数点のことをいいます．メモリ上のスカラ浮動小数点は，オペランドで示されたアドレスのメモリ上にある，一つの単精度浮動小数点，あるいは倍精度浮動小数点のことをいいます．

XMMレジスタ上のスカラの浮動小数点は，XMMレジスタ自体がパックされた浮動小数点しか格納できないため，オペランドでXMMレジスタが指定された場合のスカラは，XMMレジスタの最下位に位置する単精度浮動小数点，あるいは倍精度浮動小数点のことを指すことになります（図2）．

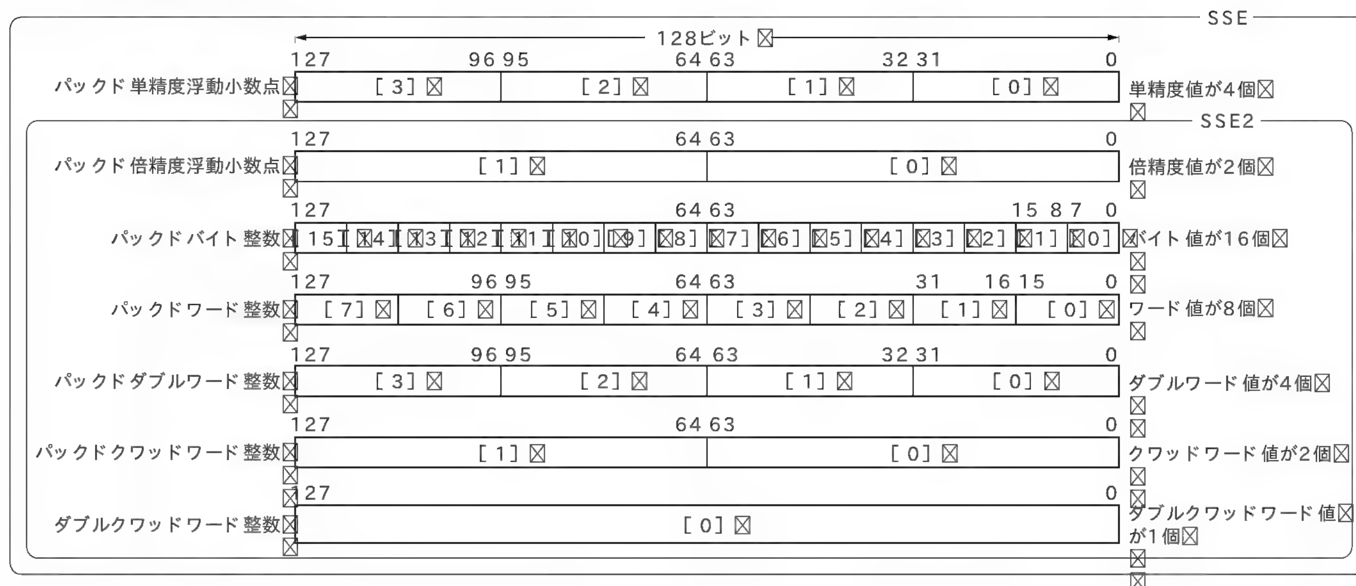
● SSE/SSE2命令で使用されるレジスタ

SSE/SSE2命令は，新規に追加された「XMMレジスタ」と「MXCSRレジスタ」の二種類のレジスタを使い実行されます．

XMMレジスタは，今述べたようにパックされた値を格納するためのレジスタです．そしてMXCSRレジスタは，SSE/SSE2命令の浮動小数点演算の制御と状態を表すためのレジスタです．XMMレジスタ，MXCSRレジスタと，CPUの他のレジスタ，メモリとの関係を図3に示します．

XMMレジスタに対する値のロード/ストアは，一つのデータ

〔図1〕SSE/SSE2命令が扱うデータ



のサイズが大きいため，メモリとのやり取りが主となります．しかし，32ビット整数値を扱う場合は，CPUの32ビット汎用レジスタ(EAX，EBX，…)ともデータのやり取りが可能です．また，64ビット以下のパックされた整数を扱う場合は，MMXレジスタともデータのやり取りを行うことができます．

ほとんどのSSE/SSE2命令は，CPUのEFLAGSレジスタに影響を与えません．しかし，SSE/SSE2の比較命令には，スカラの浮動小数点値同士の比較結果を直接，CPUのEFLAGSレジスタにセットする命令もあります．そのため，比較命令によってはCPUのEFLAGSレジスタに影響を与える場合があります．

MXCSRレジスタは，メモリに対してのみデータをロード/ストアすること可能です．これにより，メモリ経由にはなりますが，MXCSRレジスタの内容取得や設定といったことが行えます．

(1) XMMレジスタ

SSE/SSE2命令で扱うパックド単精度浮動小数点，およびパックド倍精度浮動小数点の値と，128ビット長のパックド整数の値は，128ビット長のXMMと呼ばれるレジスタに格納し，演算を行います．とくに演算の場合は，XMMレジスタがかならず転送先となります．XMMのレジスタは8本あり，各レジスタにはXMM0～XMM7という名前が付けられています(図4)．

MMX命令の場合，そこで使用されるMMXレジスタは，物理的にはFPU(浮動小数点演算ユニット)のレジスタスタックの一部のビットを借りる形で存在していました．そのため，MMX命令とFPU命令はいっしょに実行することができませんでした．しかし，XMMレジスタは物理的に独立したレジスタとして存在しています．そのため，XMMレジスタのみを使用するSSE/SSE2命令は，FPU命令といっしょに実行することが可能となっています．

(2) MXCSRレジスタ

MXCSRレジスタは，32ビット長のレジスタでSSE/SSE2命令の浮動小数点演算の制御(コントロール)と状態(ステータス)を表すためのレジスタです．MXCSRレジスタを簡単にいうと，以前FPU命令のところで説明した「コントロールレジスタ」と「ステータスレジスタ」を簡素化し，一つのレジスタにしたものといえます．また，各ビットの動作もFPU命令のコントロール/ステータスレジスタと類似しています．

MXCSRレジスタは，図5のように「例外フラグ」,「例外マスク」,「丸め制御」,「ゼロフラッシュ」の四つのフィールドから構成されています．「例外フラグ」は，FPUのステータスレジスタ,「例外マスク」と「丸め制御」の二つは，FPUのコントロールレジスタと同一の動作をします．「ゼロフラッシュ」は，SSE/SSE2命令で設けられた新しいビットです．

▶ 例外フラグ

MXCSRレジスタのビット0～5の下位6ビットは，ステータスのビットで，SSE/SSE2命令の浮動小数点演算で発生した例外を表すフラグとして使われています．フラグは，0が例外発生なし，1が例外発生を示します．

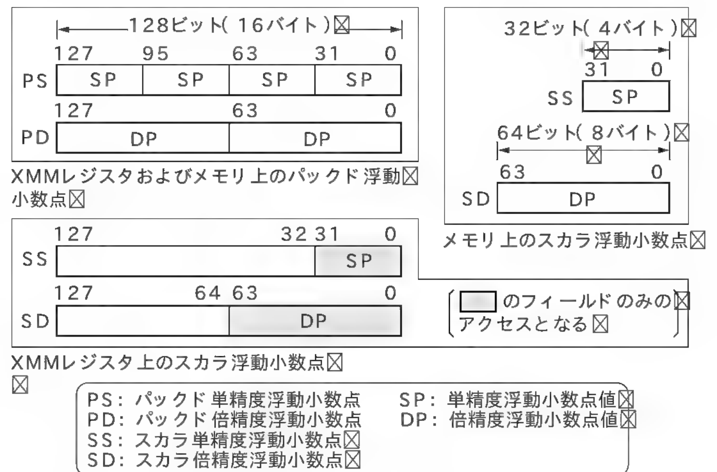
この例外フラグは，一度1にセットされると，後は自動的に0になることはありません．そのため，一度1にセットされたフラグは，プログラムにより0を書き込んでクリアする必要があります．

初期値では，例外フラグはすべて0になっています．

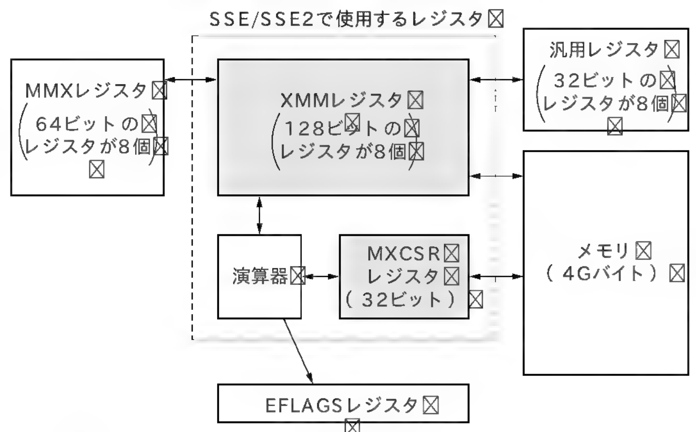
▶ 例外マスク

MXCSRレジスタのビット7～12は，コントロールのビット

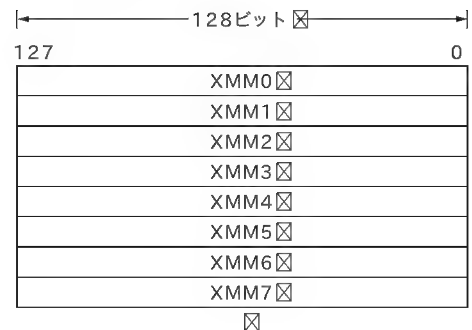
〔図2〕パックドとスカラ



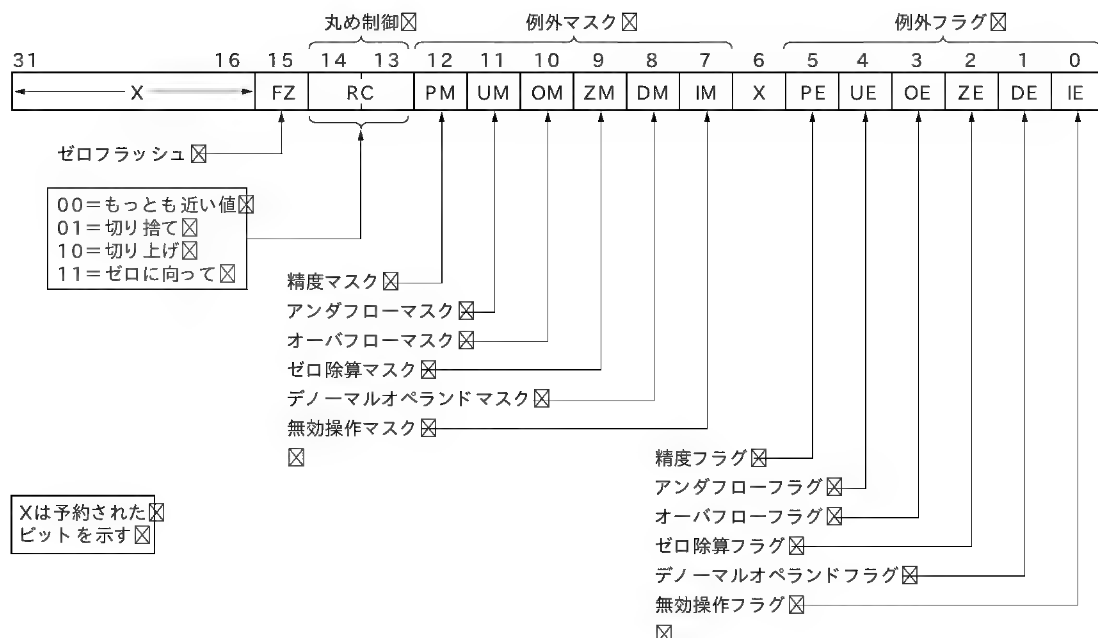

〔図3〕SSE/SSE2で使用するレジスタとメモリの関係




〔図4〕XMMレジスタ



〔図5〕MXCSRレジスタ




で，例外発生時に割り込みを発生させるか否かを指定するマスクです．マスクは，1で割り込み発生なし，0で割り込み発生ありとなります．マスクされた状態(1)で，SSE/SSE2命令を実行し，例外が発生すると，FPUと同じように発生した例外に対応した値が結果として返されます．

初期値では，例外マスクはすべて1(割り込み発生なし)になっています．

▶ 丸め制御

MXCSRレジスタのビット13，14は，演算結果や高い精度の浮動小数点を低い精度の浮動小数点や整数に変換する場合の丸めの方法を指定します．初期値では，丸め制御はもっとも近い値に設定されています．

▶ ゼロフラッシュ

このゼロフラッシュは，ゼロフラッシュモードのON/OFFを制御するビットです．0でOFF，1でONとなります．初期値では，ゼロフラッシュは0(OFF)になっています．

このゼロフラッシュモードは，アンダフローの例外がマスク(1)されている場合の，CPUの動作を変更するものです．そのため，アンダフロー例外のマスクが0だった場合は，このゼロフラッシュのビットは意味をもちません．

ゼロフラッシュモードがOFF，アンダフローの例外がマスク(1)された状態で，アンダフロー例外が発生すると，FPU命令と同じようにSSE/SSE2命令も結果としてデノーマルな値を返します．

しかし，ゼロフラッシュモードがONで，アンダフローの例外がマスク(1)されていた状態で，アンダフロー例外が発生すると，SSE/SSE2命令は結果として符号付きのゼロを返してきます．

このゼロフラッシュモードは，頻繁にアンダフローが発生する状況で，アンダフローの結果をゼロと見なしてもよい場合に，演算の速度を上げるために使用されます．ただし，アンダフローの結果をゼロとしてしまうので，精度の点で多少の低下が発生します．

● SSE/SSE2命令の使用上の注意

ここでは，SSE/SSE2命令を使用するに当たり注意する必要がある事がらについて説明します．

(1) メモリ上の128ビットデータは，アドレスが16バイトの倍数になっている必要がある

これは，SSE/SSE2命令を使用するうえで，一番重要な事がらといえます．SSE/SSE2命令は，128ビット，16バイトという大きな値を扱うため，メモリアクセス機構の関係からか，メモリ上の128ビットデータは，アドレスが16バイトの倍数の位置に配置されている必要があります．

もし，メモリ上の128ビットデータにアクセスするとき，アドレスが16バイトの倍数でないと一般保護例外が発生してしまいます．そのため，プログラム上で128ビットデータをディレクティブで確保する場合，MASMなら「ALIGN 16」，gasなら「.align 16」を使い，事前にアライメントを16バイトの倍数にする必要があります．

(2) スタックを使い128ビットデータを引き数としてサブルーチンに渡すときには？

今述べたようにSSE/SSE2命令は，アドレスが16バイトの倍数でないと128ビットデータのアクセスはできません．しかしスタックを使い128ビットデータを引き数としてサブルーチンに渡す場合などは，この「アドレスが16バイトの倍数」という制限は，現実的に難しい面があります．

そこで，アドレスが16バイトの倍数でない128ビットデータのアクセスには，MOVUPS，MOVUPD，MOVDQUというニモニッ

クにUの付いた特別な命令を使用し，メモリをアクセスすることになります．この三つの命令は，他のSSE/SSE2命令とは異なり，アクセスするメモリ上の128ビットデータのアドレスが16バイトの倍数でなくても例外を発生しません．

ただし，アドレスが16バイトの倍数でない128ビットデータのアクセスは，多少実行速度の点で不利なので，必要がなければMOVUPS，MOVUPD，MOVDQUは，あまり多用しないほうが良いでしょう．

（3）MXCSRレジスタのビット変更

MXCSRレジスタは，先に述べたように制御と状態表示を一つのレジスタで行っています．そのため，MXCSRレジスタのビット変更は，必要のないビットを変化させないように注意する必要があります．

具体的には，STMXCSR命令で，MXCSRレジスタの内容をメモリにストアします．その後，メモリ上のMXCSRレジスタの内容に対して，ANDやOR命令で必要なビットのセット/クリアを行います．その上で，LDMXCSR命令で変更済みのメモリ上の新しいMXCSRレジスタの内容を，実際のMXCSRレジスタにロードします．

（4）MMXレジスタをアクセスするSSE/SSE2命令を使用した場合の注意

SSE/SSE2命令の中には，転送元あるいは転送先のオペランドにMMXレジスタが指定されるものがあります．そのような命令では，MMX命令のときと同じようなFPUのレジスタスタックからMMXレジスタへの切り替えが行われます．このとき，FPUのレジスタスタックが空でないと，正しくMMXレジスタへ切り替わりません．

そのため，オペランドにMMXレジスタが指定されるSSE/SSE2命令を使用する場合は，すでにMMXレジスタが有効になっているか，あるいはFPUのレジスタスタックが有効なら，かならずレジスタスタックが空の状態で実行する必要があります．

また，今述べたようにオペランドにMMXレジスタが指定されるSSE/SSE2命令を実行すると，MMXレジスタが有効になるので，FPUのレジスタスタックに戻す場合は，MMX命令のうちのEMMS命令を実行する必要があります．

*　　　　*

次回はSSE/SSE2の各命令の動作について解説します．

おおぬき・ひろゆき　大貫ソフトウェア設計事務所

TMS320C6713搭載DSPスタータキットを使った

# C++によるDSPオブジェクト指向プログラミング

## 第2回 アナログ信号入出力用クラスを使う簡単なプログラム(前編)

◆三上 直樹

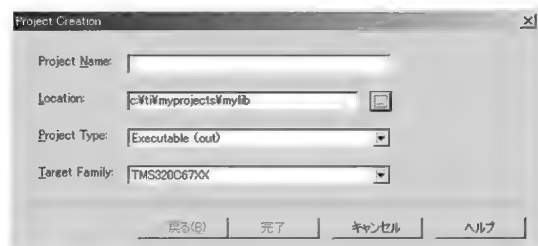
前回は，TMS320C6713 DSPスタータキット(以下，C6713 DSK)の，アナログ信号の入出力で使うクラスのソースプログラムを作成しました．今回は，まずこれをライブラリにする方法を説明します．次に，このライブラリを使ってアナログ信号の入出力を行う簡単なプログラムを示します．

### 1 ライブラリの作成

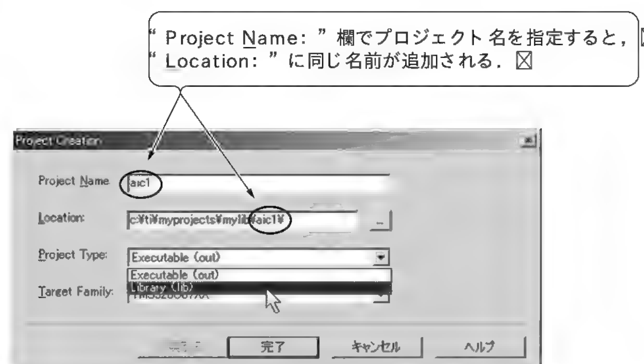
CCS(Code Composer Studio)注1では，ライブラリや実行プログラムの作成をプロジェクトという単位で行います．つまり，プロジェクトごとに一つのフォルダ(ディレクトリ)を作り，関連するいくつかのファイルをそのフォルダに置きます．

それでは，ライブラリを作成する手順を新規プロジェクトの作成から順に示していきます．

〔図1〕ライブラリを作成する際のプロジェクト設定のようす




(a)"Location"に"C:¥ti¥myprojects¥mylib"を指定




(b)プロジェクト名を"aic1"に指定すると…

● ライブラリ作成のための新規プロジェクトの作成

CCSを立ち上げ，メニューバーから[Project | New...]を選択すると図1(a)のようなウィンドウが開きます．"Location:"の欄でこれから作成するプロジェクトを格納するドライブ名とフォルダ名を指定します注2．ここ(図1)では"c:¥ti¥myprojects¥mylib"を指定しました．次に"Project Name:"の欄でプロジェクト名を指定します．プロジェクト名を"aic1"と指定すると，図1(b)のように"Location:"の欄が"c:¥ti¥myprojects¥mylib¥aic1¥"に変わります．これから作成するプロジェクトに関するファイル一式はこのフォルダに入ることになります．

"Project Type:"の欄は，デフォルトで図1(a)のように"Executable(out)"となっており，そのままでは実行プログラムを作るようになります．ここではライブラリを作るので，図1(b)のように"Library(lib)"を選択します．

"Target Family:"の欄は，デフォルトで図1(a)のように"TMS320C67XX"となっています．TMS320C6713用のライブラリを作る場合はそのままにしておきます．

● 新規プロジェクトのオプション設定

オプションを設定する前に，作成するコードをDebug版にするかRelease版にするかを決めます．デフォルトではDebug版を作成するように設定されていますが，ここではRelease版のコードを生成することにします．そこで，図2に示すように，ツールバーのDebugと表示されたドロップダウンコンボボックスで"Release"注3を選択します．

次に，プロジェクトをビルドする際のオプションを設定します．ここではコンパイラのオプションとアーカイバのオプションを設定します．この設定を行うためには，メニューバーから[Project | Build Options…]を選択します．

コンパイラオプションの設定では"Compiler"タブを選択します．ここでは"Category:"欄の"Basic"という項目を選択し，"Target Version:"のドロップダウンコンボボックスから図3(a)のように"C671x(-mv6710)"を選択します．

注1：以下ではCCSのバージョン2.2に基づいて説明を行う．
注2：右側のボタンで，ブラウズして選択することもできる．
注3：同じことは，ビルドの際のオプション設定でも行うことができる．ただし，その場合はいくつかの項目の設定が必要になる．

〔図2〕Debug版とRelease版の切り替え




アーカイバのオプション設定では，"Archiver"タブを選択します．デフォルトでは図3(b)のように，作成されるライブラリの名前がプロジェクト名(この場合は"aic1.lib")になるので，必要に応じて変更します．ここでは"aic23.lib"という名前に変更します．

● ライブラリのビルド

新規プロジェクトのオプション設定が終わったら，次にライブラリを作るために必要なソースファイルをプロジェクトに追加します．この操作はメニューバーからも行えますが，マイコンピュータなどでソースファイルが存在するフォルダを開き，ファイルを選択し，CCSの"Project view"ウィンドウの"Source"へドラッグ&ドロップすることでも追加できます．ここで必要な三つのソースファイルAIC23_Base.cpp，AIC23_Intr.cpp，InitC6713DSK.cppを追加したところを図4に示します．

必要なソースファイルが追加されたところで，ビルドを行います．メニューバーから[Project | Build]を選択するか，ボタンをクリックすればビルドが開始します．エラーがなければ，プロジェクトの入ったフォルダの下の"Release"フォルダにaic23.libというライブラリが作成されます．

作成されたライブラリは必要に応じてしかるべきフォルダにコピーまたは移動しておきます．筆者は"myprojects"の下に"lib"というフォルダを作り，そこへライブラリファイルaic23.libを置いています．また，"myprojects"の下に"include"というフォルダを作り，そこへこのライブラリに関連する四つのヘッダファイルAIC23_Base.hpp，AIC23_Polling.hpp，AIC23_Intr.hpp，InitC6713DSK.hppを置いています．

なお，ヘッダファイルを置く"myprojects¥include"には，デフォルトでパス(path)が設定されていません．その場合，プロジェクトを作るたびにビルドのオプションでパスを設定しなければならないため，手間がかかります．そこで，"myprojects¥include"へのパスを通しておいたほうがよいでしょう．この方法はp.152のコラム"インクルードファイルのパスの指定"を参照してください．

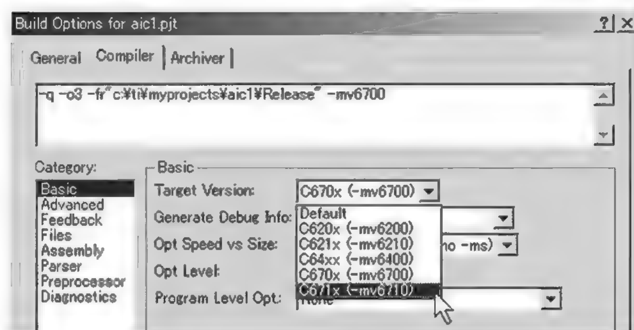
## 2 実行プログラムの作成方法

● 実行プログラム作成のための新規プロジェクト作成

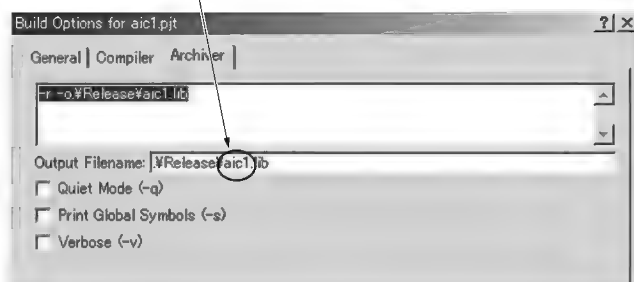
実行プログラムを新規に作成する際は，ライブラリの新規作成と同様に，メニューバーから[Project | New...]を選択します．図1(a)のようなウィンドウが開いたら，"Location:"と"Project Name:"の欄を設定し，プロジェクト用のフォルダ名を指定します．"Project Type:"の欄はデフォルトで"Executable(.out)"となっているので，そのままにします．

次に，ツールバーで"Debug"か"Release"を選択した後，メニューバーで[Project | Build Options...]でコンパイラオプションを設定するため，図3(a)のように"C671x(-mv6710)"を選択します．最後に，必要なファイルをプロジェクトに追加します．実行プログラムを作成する場合は，main()関数を記述しているソースファイルの他に最低限，次に示す二つのファイル

〔図3〕ライブラリ作成の際のビルドオプションの設定のようす




(a)"C671x(-mv6710)"を選択




(b)作成されるライブラリの名前がプロジェクト名になる

〔図4〕ライブラリを作るために必要なソースファイルをプロジェクトに追加したようす



が必要になります．一つはリセット/割り込みに対応するベクタを記述したアセンブリ言語によるファイルです．もう一つはリンカで使うリンカコマンドファイルです．これらのファイルは自分で作成する必要があります．

図5には次項で作る継承を使わない素朴なプログラムの実行プログラムを作成するために必要となるファイルをすべて追加したようすを示します．

● リセット/割り込みベクタ用アセンブラファイル[注4]

リスト1にリセットベクタを記述したファイル(vects_Reset.asm)を示します．_c_int00はC/C++のエントリポイントとして予約されているシンボルです．

最初に，MVKLとMVKHの二つの命令でレジスタA0にエントリポイントのアドレスを設定し，そのアドレスへB命令で分岐します．B命令の後にはNOP命令が5個並んでいますが，これはB命令が遅延分岐(delayed branch)命令になっており，その遅延スロットの数が5になっているからです．このNOP命令を置かない場合，分岐先の命令を実行する前に余計な命令が実行され，正常に動作しない場合があります．

以降では，割り込みを使用しない場合には，基本的にこのファイルを使うことにします．

● リンカコマンドファイル

リスト2にリンカコマンドファイル(lnk_std.cmd)を示します．

-stackはスタックのサイズを指定するオプションです．スタックの領域は，ローカル変数を割り当てたり，関数の引き数や戻り値を保管したりという用途で使います．このサイズのデフォルト値は0x400(1024)バイトですが，ここでは余裕を持たせるためサイズの指定を0x1000(4096)バイトにしています．

注4：ここで示すリストは割り込みを使用しない場合に使うことを想定しているので，割り込みベクタに関する部分は記述していない．割り込みベクタに関する部分を含んだものは次回に解説する．

## Column インクルードファイルのパスの指定

ヘッダファイルなどのインクルードファイルが現在作成中のプロジェクトのフォルダ以外に置かれている場合は，インクルードする際にそのフォルダ名もいっしょに指定するか，ビルドのオプションでパス(path)を設定する必要があります．ただし，ここで作成するライブラリのためのヘッダファイルのように汎用的なものでは，新しいプロジェクトを作るたびにそれを行う必要があるため，めんどうです．そこで，筆者はCode Composer Studioに関係するレジストリに，パスを設定するためのキーを追加しています．そのようすを図Aに示します．

キーの追加は次のように行います．ルートキー“HKEY_LOCAL_MACHINE”の下の“SOFTWARE¥Texas Instruments¥CCS_c:|ti|¥TMS320C67XX¥Build Tools¥Compiler”の下にSearchPath4[注A]というキーを追加します．そして，このキーの項目としては，“bActive”，“Order”，“Path”を作成します．

“bActive”の値は上に設定します．“Order”の値は，すでに1～3まで設定されているので，その次の値である4にします．“Path”はインクルードファイルの置かれているフォルダ名を設定します．

なお，以上の設定はWindows 2000の場合で，他のOSの場合はここでの説明と異なる場合があるので，注意してください．また，この設定に関しては特に公表されたドキュメントがあるわけではないので，正しい方法だという保証はありません．

同じ方法で，ライブラリのパスも追加できるかどうかを確かめてみました．“…¥Build Tools¥Linker”の下に同様のキーを追加してみましたが，こちらのほうはリンクの際にライブラリファイルをリンクすることは失敗しました．

ライブラリファイルのほうは一度リンカコマンドファイルにフルパス名で追加しておけば済むことなので，特にライブラリのためのパスを設定しておかなくても，それほど不便ではありません．

注A：すでにキーがSearchPath3まで作成されていたので，SearchPath4という名前を付けた．

〔図A〕レジストリエディタによってインクルードファイルのパスを追加するようす



-heapはヒープ領域のサイズで，C++の演算子newで動的に確保した領域として使います．こちらもデフォルトのサイズは0x400(1024)バイトです．しかし，動的メモリの割り当てを行う際に大きな領域が必要になる場合もあるので，この連載ではサイズの指定を0x4000(16384)バイトにしています．

-lはリンクするライブラリを指定するためのオプションです．rts6700.libはCCSのC/C++コンパイラで標準的にサポートされるTMS320C67xx用のランタイムライブラリです．csl6713.libはチップサポートライブラリで，dsk6713bsl.libはC6713DSK用のボードサポートライブラリです．c:¥ti¥myprojects¥lib¥AIC23.libは前項で作成したライブラリです．

MEMORYの部分には，ターゲットのメモリ配置を指定します．TMS320C6713に内蔵するRAMは0x0000 0000番地から割り当てられています．その先頭はリセット/割り込みのベクタ用に予約されているため，その領域にはvectsという名前を付けています．内蔵RAMの残りの領域にはIRAMという名前を付けています．また0x8000 0000から配置されている外部RAMに対してはSDRAMという名前を付けています．

SECTIONの部分では，データや命令コードをMEMORYの部分で名前を付けたどの領域に配置するのかを指定します．主としてよく使うデータはIRAMに，初期化のみに使うようなデータなどと命令コードはSDRAMに配置するように指定しています．

なお，SECTIONの部分で最初に現れるvectorsは，リスト1の.sect "vectors"に対応します．したがって，この部分は同じ名前にする必要があります．

以降では，スタック領域やヒープ領域のサイズを変更する必要がないかぎり，基本的にこのファイルを使うことにします．

## 3 ポーリング方式でアナログ信号入出力を行う簡単なプログラム

それでは，作成したライブラリを使ってプログラムを作ります．最初は簡単なプログラムで，A-D変換器から入力されたデータを，何も処理せずにそのままD-A変換器に送るというプログラムを作ります．このプログラムを二つの実現方法で作ります．一つ目はクラスの継承を使わない素朴なプログラムです．二つ目は，AIC23_Pollingクラスを継承する派生クラスを作るという方法を使うプログラムです．

### ● 継承を使わない素朴なプログラム

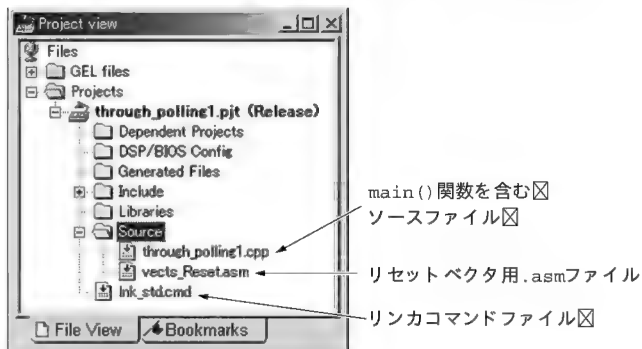
リスト3にプログラム(through_polling1.cpp)を示します．このプログラムではポーリング方式を使うので，AIC23_Polling.hppをインクルードします．

メインプログラムでは最初にAIC23_Pollingクラスのオブジェクトdskを宣言しています．これにより，AIC23_Pollingクラスのコンストラクタが起動し，DSKボードの初期化などが行われます．

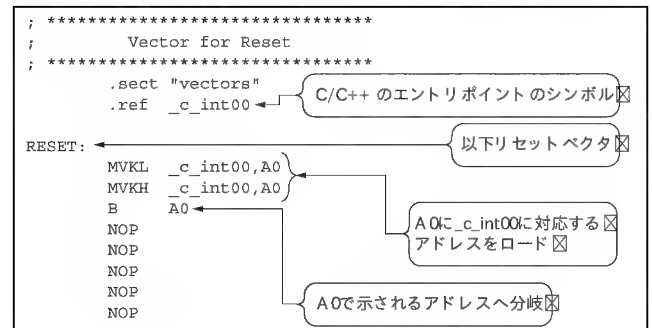
このとき，ただ単にdskと記述すると，コンストラクタで指定しているデフォルトの引き数の値が設定されます．したがって，このプログラムの場合は，標本化周波数が48kHzに，2次キャッシュのサイズが64Kバイトに設定されます．

AIC23_Polling dsk;以下のコメントにしている6行は，

〔図5〕「継承を使わない素朴なプログラム」の実行ファイル作成に必要なすべてのファイルをプロジェクトに追加したようす




〔リスト1〕アセンブリ言語によるリセットベクタに関する記述(vects_Reset.asm)

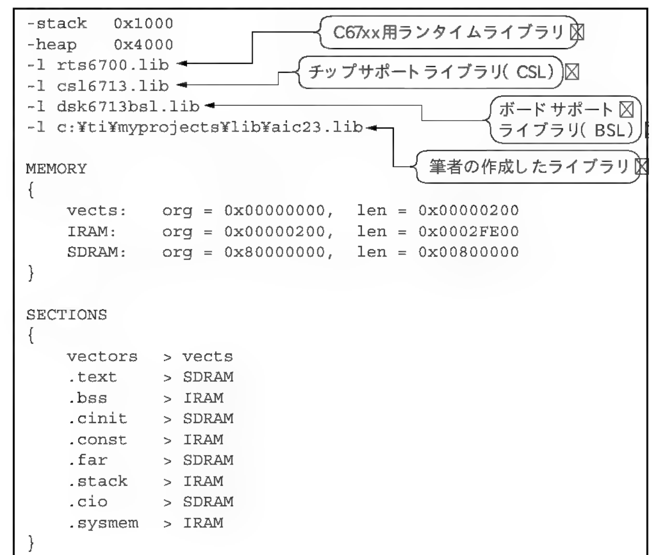


```
; *********************************
;          Vector for Reset
; *********************************
        .sect "vectors"
        .ref  _c_int00

RESET:
        MVKL  _c_int00,A0
        MVKH  _c_int00,A0
        B     A0
        NOP
        NOP
        NOP
        NOP
        NOP
```



〔リスト2〕リンカコマンドファイル(lnk_std.cmd)



```
-stack   0x1000
-heap    0x4000
-l rts6700.lib
-l csl6713.lib
-l dsk6713bsl.lib
-l c:¥ti¥myprojects¥lib¥aic23.lib

MEMORY
{
    vects:    org = 0x00000000,  len = 0x00000200
    IRAM:     org = 0x00000200,  len = 0x0002FE00
    SDRAM:    org = 0x80000000,  len = 0x00800000
}

SECTIONS
{
    vectors   > vects
    .text     > SDRAM
    .bss      > IRAM
    .cinit    > SDRAM
    .const    > IRAM
    .far      > SDRAM
    .stack    > IRAM
    .cio      > SDRAM
    .sysmem   > IRAM
}
```

〔リスト 3〕ポーリング方式による A-D 変換されたデータをそのまま D-A 変換するプログラム ―― 継承を使わない素朴な方法 (through_polling1.cpp)

```
// ------------------------------------------------------------
//    ポーリング方式により，A-D 変換されたデータをそのまま D-A 変換する
//         ( 素朴な方法)
// ------------------------------------------------------------

#include "AIC23_Polling.hpp"

int main()
{
    AIC23_Polling dsk;
      //AIC23_Polling dsk(AIC23_Base::fs48kHz);
      //AIC23_Polling dsk(AIC23_Polling::fs48kHz);
      //AIC23_Polling dsk(dsk.fs48kHz);
      //AIC23_Polling dsk(AIC23_Polling::fs48kHz,
      //                                InitC6713DSK::cache64K);
      //AIC23_Polling dsk(dsk.fs48kHz, InitC6713DSK::cache64K);
    short ch[2];

    while( 1)
    {
        dsk.Read( ch) ;
        dsk.Write( ch) ;
    }
}
```

〔リスト 4〕ポーリング方式による A-D 変換されたデータをそのまま D-A 変換するプログラム ―― 継承を使う方法 (through_polling2.cpp)

```
//------------------------------------------------------------
//  ポーリング方式により，A-D 変換されたデータをそのまま D-A 変換する
//         ( 継承を使う方法)
//------------------------------------------------------------

#include "AIC23_Polling.hpp"

class ThroughPolling : public AIC23_Polling
{
public:
    ThroughPolling(const AICfs fs_set = fs48kHz,
                   const L2mode banks = cache64K)
                 : AIC23_Polling(fs_set, banks) {}
    void Execute();
private:
    short ch[2];
};

void ThroughPolling::Execute()
{
    Read(ch);
    Write(ch);
}

int main()
{
    ThroughPolling dsk;

    while(1) dsk.Execute();
}
```

コンストラクタに引き数を与える場合の例です．このとき注意すべき点は，与える引き数がクラスのメンバとして定義されている enum 型の定数であるということです．

標本化周波数を設定するための定数は AIC23_Base クラスの中で定義されているので，単に 48kHz に対応する定数 fs48kHz をコンストラクタの引き数として記述するとコンパイルエラーになります．そこで，一つの書き方として，定数 fs48kHz の頭に AIC23_Base:: を付けるという方法があります．また，AIC23_Polling は AIC23_Base を公開(private) 継承する派生クラスになっているので，AIC23_Polling:: または dsk. を付け，AIC23_Polling::fs48kHz，または dsk.fs48kHz のように記述してもかまいません．

2次キャッシュのサイズは InitC6713DSK クラスの中で定義されているので，64K バイトに対応する定数である cache64K の頭に InitC6713DSK:: を付け，InitC6713DSK::cache64K のように記述します．2次キャッシュのサイズの場合に dsk.cache64K などのように記述できない理由は，AIC23_Polling クラスの基底クラスである AIC23_Base クラスが InitC6713DSK クラスを限定公開(protected) 継承する派生クラスになっているからです．

while のループでは，dsk.Read(ch) で TLV320AIC23 の A-D 変換器から送られてきた 2チャネルのデータを読み込み，そのデータを何も処理せずそのまま dsk.Write(ch) で TLV320AIC23 の D-A 変換器へ送るという処理を繰り返します．

● 継承を使うプログラム

リスト 3と同じ内容の処理を AIC23_Polling クラスを継承する派生クラスを作って実現するプログラムがリスト 4 (through_polling2.cpp) です．

ThroughPolling クラスは AIC23_Polling クラスを公開継承する派生クラスです．公開メンバとして，コンストラクタ ThroughPolling() とメンバ関数 Execute() を宣言します．Execute() の処理の内容は，Read(ch) でデータを読み込み，何も処理をせずそのまま Write(ch) でデータを送り出すという処理です．

データは非公開メンバとして宣言されています．

メインプログラムでは最初に ThroughPolling クラスのオブジェクト dsk を宣言しています．この記述はリスト 3と同様にコンストラクタでデフォルト引き数の値が設定されます．引き数を与える場合はリスト 3と同様になるので，その書き方の例は省略します．

while のループでは，dsk.Execute() を繰り返すだけの処理になっています．

● まとめ

以上，同じ処理内容を実現する，スタイルの異なった二つのプログラムを紹介しました．オブジェクト指向ということを考慮すると，以降のプログラムもリスト 4のスタイルで記述した方がよいのかもしれません．しかし，特にこのようなスタイルをとるメリットが見つからないばかりか，逆にリストが長くなりプログラムが理解しにくくなるのではないでしょうか．したがって，以降では，基本的にリスト 3のスタイルでプログラムを記述することにします．

次回は，割り込みを使ったアナログ信号入出力と FIR フィルタのプログラムを示します．

みかみ・なおき　職業能力開発総合大学校 情報工学科

# やり直しのための信号数学 第21回

## DCTの高速計算アルゴリズム

三谷 政昭



前回（2004年2月号）は，「DCT，IDCTの効率的構成法」と題し，ローパスフィルタとハイパスフィルタを直並列的に配置したフィルタバンクの“ツリー構成”と呼ばれるシステムを紹介した．

今回は，フーリエ変換と密接な関係を有するDCTの高速実現法として演算量の少ない計算アルゴリズムを説明する．具体的には，FFT（高速フーリエ変換）を利用する方法とDCTの係数行列を分解する方法をとり上げて，基本的な考え方を中心に説明する．（筆者）

### DCTの演算量

まず，$N$個のディジタル信号のサンプル値$\{x_k\}_{k=0}^{k=N-1}$に対して，DCT値$\{C_\ell^{(N)}\}_{\ell=0}^{\ell=N-1}$を算出する式を示す．

$$C_\ell^{(N)}=\frac{1}{N}\sum_{n=0}^{N-1}\gamma_\ell x_n\cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} \quad\cdots\cdots(1)$$

$$\text{ただし，}\gamma_\ell=\begin{cases}1 & ;\ell=0\\ \sqrt{2} & ;\ell\neq 0\end{cases} \quad\cdots\cdots(2)$$

たとえば，4サンプルのディジタル信号$\{x_0, x_1, x_2, x_3\}$に対するDCT値$\{C_\ell^{(4)}\}_{\ell=0}^{\ell=3}$の計算式を書き下してみよう．式(1)，式(2)に$N=4$を代入すればよい．

$$\begin{cases}
C_0^{(4)}=\dfrac{1}{4}\{x_0+x_1+x_2+x_3\}\\
C_1^{(4)}=\dfrac{\sqrt{2}}{4}\left\{x_0\cos\left(\dfrac{\pi}{8}\right)+x_1\cos\left(\dfrac{3\pi}{8}\right)+x_2\cos\left(\dfrac{5\pi}{8}\right)+x_3\cos\left(\dfrac{7\pi}{8}\right)\right\}\\
C_2^{(4)}=\dfrac{\sqrt{2}}{4}\left\{x_0\cos\left(\dfrac{2\pi}{8}\right)+x_1\cos\left(\dfrac{6\pi}{8}\right)+x_2\cos\left(\dfrac{10\pi}{8}\right)+x_3\cos\left(\dfrac{14\pi}{8}\right)\right\}\\
C_3^{(4)}=\dfrac{\sqrt{2}}{4}\left\{x_0\cos\left(\dfrac{3\pi}{8}\right)+x_1\cos\left(\dfrac{9\pi}{8}\right)+x_2\cos\left(\dfrac{15\pi}{8}\right)+x_3\cos\left(\dfrac{21\pi}{8}\right)\right\}
\end{cases} \quad\cdots\cdots(3)$$

ところで，式(1)のDCT計算式において，$\ell$行$n$列の要素として，

$$\lambda_{\ell n}^{(N)}=\gamma_\ell\cos\left\{\frac{(2n+1)\ell}{2N}\pi\right\} \quad\cdots\cdots(4)$$

$$\ell,\ n=0,\ 1,\ 2,\ \cdots,(N-1)$$

とおけば，DCT計算は一般的な行列演算の形式で表現できる．なお，行列のサイズは$N\times N$（$N$行$N$列の意味）である．少しわかりにくいかもしれないので，以下に式(3)の行列表現を示しておくことにする．

$$\begin{pmatrix}C_0^{(4)}\\C_1^{(4)}\\C_2^{(4)}\\C_3^{(4)}\end{pmatrix}=\frac{1}{4}\begin{pmatrix}\lambda_{00}^{(4)}&\lambda_{01}^{(4)}&\lambda_{02}^{(4)}&\lambda_{03}^{(4)}\\\lambda_{10}^{(4)}&\lambda_{11}^{(4)}&\lambda_{12}^{(4)}&\lambda_{13}^{(4)}\\\lambda_{20}^{(4)}&\lambda_{21}^{(4)}&\lambda_{22}^{(4)}&\lambda_{23}^{(4)}\\\lambda_{30}^{(4)}&\lambda_{31}^{(4)}&\lambda_{32}^{(4)}&\lambda_{33}^{(4)}\end{pmatrix}\begin{pmatrix}x_0\\x_1\\x_2\\x_3\end{pmatrix} \quad\cdots(5)$$

$$\begin{cases}
\lambda_{00}^{(4)}=\lambda_{01}^{(4)}=\lambda_{02}^{(4)}=\lambda_{03}^{(4)}=1,\\
\lambda_{10}^{(4)}=\sqrt{2}\cos\left(\dfrac{\pi}{8}\right),\ \lambda_{11}^{(4)}=\sqrt{2}\cos\left(\dfrac{3\pi}{8}\right),\\
\lambda_{12}^{(4)}=\sqrt{2}\cos\left(\dfrac{5\pi}{8}\right),\ \lambda_{13}^{(4)}=\sqrt{2}\cos\left(\dfrac{7\pi}{8}\right),\\
\lambda_{20}^{(4)}=\sqrt{2}\cos\left(\dfrac{2\pi}{8}\right),\ \lambda_{21}^{(4)}=\sqrt{2}\cos\left(\dfrac{6\pi}{8}\right),\\
\lambda_{22}^{(4)}=\sqrt{2}\cos\left(\dfrac{10\pi}{8}\right),\ \lambda_{23}^{(4)}=\sqrt{2}\cos\left(\dfrac{14\pi}{8}\right),\\
\lambda_{30}^{(4)}=\sqrt{2}\cos\left(\dfrac{3\pi}{8}\right),\ \lambda_{31}^{(4)}=\sqrt{2}\cos\left(\dfrac{9\pi}{8}\right),\\
\lambda_{32}^{(4)}=\sqrt{2}\cos\left(\dfrac{15\pi}{8}\right),\ \lambda_{33}^{(4)}=\sqrt{2}\cos\left(\dfrac{21\pi}{8}\right)
\end{cases} \quad\cdots(6)$$

ただし，$\lambda_\ell=\{\lambda_{\ell n}{}^{(N)}\}_{n=0}^{n=N-1}$（$\ell$ =0，1，2，…，($N$－1)）は正規直交基底ベクトルを構成するものである．この正規直交基底ベクトルから成る行列要素$\{\lambda_{\ell n}{}^{(N)}\}_{\ell,n=0}^{\ell,n=N-1}$をあらかじめ計算しておけば，式(5)からDCT計算は一般的な行列演算によって実現できることがわかる．この計算で要求される演算量は，実数乗算が4×4＝16回，実数加算が(4－1)×4＝12回である．よって，一般的な$N$サンプルの入力信号$\{x_k\}_{k=0}^{k=N-1}$に対するDCT計算では，

$$\begin{cases}\text{実数乗算}=N\times N=N^2\ \text{回}\\ \text{実数加算}=(N-1)\times N\ \text{回}\end{cases}\quad\cdots\cdots(7)$$

の演算量を要することになる．たとえば，$N$＝1000とすると，実数乗算は100万回，実数加算は90万回を要するわけであるが，これらの演算量を劇的に減らせる高速計算アルゴリズムがいろいろと提案されているので，以下に代表的な手法を紹介する．

## FFTによる高速計算アルゴリズム

FFT(高速フーリエ変換)による高速処理は，DCTとDFTの相互関係を利用するものである．

それでは，$N$＝4サンプルに対するDCT計算[式(3)]において，$\ell$＝1に対するDCT値$C_1^{(4)}$，すなわち，

$$C_1^{(4)}=\frac{1}{4}\left[x_0\left\{\sqrt{2}\cos\left(\frac{\pi}{8}\right)\right\}+x_1\left\{\sqrt{2}\cos\left(\frac{3\pi}{8}\right)\right\}+x_2\left\{\sqrt{2}\cos\left(\frac{5\pi}{8}\right)\right\}+x_3\left\{\sqrt{2}\cos\left(\frac{7\pi}{8}\right)\right\}\right]\quad\cdots(8)$$

の計算を例に採り，具体的に説明してみよう．なお，{　}は，正規直交関数系を構成する基底ベクトルの要素であることを表している．

まず，オイラーの公式，すなわち，

$$e^{j\theta}+e^{-j\theta}=2\cos\theta$$

となる関係より，

$$\cos\theta=\frac{e^{j\theta}+e^{-j\theta}}{2}=\frac{e^{-j\theta}+e^{j\theta}}{2}\quad\cdots\cdots(9)$$

と表される．たとえば，

$$\begin{cases}\cos\left(\frac{\pi}{8}\right)=\frac{e^{-j\frac{\pi}{8}}+e^{j\frac{\pi}{8}}}{2}\\ \cos\left(\frac{3\pi}{8}\right)=\frac{e^{-j\frac{3\pi}{8}}+e^{j\frac{3\pi}{8}}}{2}\\ \quad\vdots\end{cases}\quad\cdots\cdots(10)$$

というぐあいである．ここで，DFTの回転因子として，

$$W_8=e^{-j\frac{2\pi}{8}}\quad\cdots\cdots(11)$$

とおけば，式(10)は$W_8$を用いて，

$$\begin{cases}\cos\left(\frac{\pi}{8}\right)=\frac{e^{-j\frac{\pi}{8}}+e^{j\frac{\pi}{8}}}{2}=\frac{W_8^{\frac{1}{2}}+W_8^{-\frac{1}{2}}}{2}\\ \cos\left(\frac{3\pi}{8}\right)=\frac{e^{-j\frac{3\pi}{8}}+e^{j\frac{3\pi}{8}}}{2}=\frac{W_8^{\frac{3}{2}}+W_8^{-\frac{3}{2}}}{2}\\ \quad\vdots\end{cases}\quad\cdots(12)$$

と表される．よって，式(8)は，

$$C_1^{(4)}=\frac{\sqrt{2}}{8}\left\{x_0\left(W_8^{\frac{1}{2}}+W_8^{-\frac{1}{2}}\right)+x_1\left(W_8^{\frac{3}{2}}+W_8^{-\frac{3}{2}}\right)+x_2\left(W_8^{\frac{5}{2}}+W_8^{-\frac{5}{2}}\right)+x_3\left(W_8^{\frac{7}{2}}+W_8^{-\frac{7}{2}}\right)\right\}\quad\cdots\cdots(13)$$

となる．ここで，

$$W_8^{\,8}=\left(e^{-j\frac{2\pi}{8}}\right)^8=e^{-j2\pi}=\underbrace{\cos(2\pi)}_{1}-j\underbrace{\sin(2\pi)}_{0}=1\quad\cdots\cdots(14)$$

であることから，

$$\begin{cases}W_8^{-\frac{7}{2}}=W_8^{-\frac{7}{2}}\times W_8^{\,8}=W_8^{-\frac{7}{2}+8}=W_8^{\frac{9}{2}}\\ W_8^{-\frac{5}{2}}=W_8^{-\frac{5}{2}}\times W_8^{\,8}=W_8^{-\frac{5}{2}+8}=W_8^{\frac{11}{2}}\\ W_8^{-\frac{3}{2}}=W_8^{-\frac{3}{2}}\times W_8^{\,8}=W_8^{-\frac{3}{2}+8}=W_8^{\frac{13}{2}}\\ W_8^{-\frac{1}{2}}=W_8^{-\frac{1}{2}}\times W_8^{\,8}=W_8^{-\frac{1}{2}+8}=W_8^{\frac{15}{2}}\end{cases}\quad\cdots(15)$$

となる関係が成立する．よって，式(13)は次のように変形できる．

$$C_1^{(4)}=\frac{\sqrt{2}}{8}\left\{x_0\left(W_8^{\frac{1}{2}}+W_8^{\frac{15}{2}}\right)+x_1\left(W_8^{\frac{3}{2}}+W_8^{\frac{13}{2}}\right)+x_2\left(W_8^{\frac{5}{2}}+W_8^{\frac{11}{2}}\right)+x_3\left(W_8^{\frac{7}{2}}+W_8^{\frac{9}{2}}\right)\right\}\quad\cdots(16)$$

続けて，$W_8^{\frac{1}{2}}$をくくり出し，整理すると，

$$\begin{aligned}C_1^{(4)}&=\frac{\sqrt{2}}{8}W_8^{\frac{1}{2}}\left\{x_0\left(1+W_8^{\,7}\right)+x_1\left(W_8+W_8^{\,6}\right)+x_2\left(W_8^{\,2}+W_8^{\,5}\right)+x_3\left(W_8^{\,3}+W_8^{\,4}\right)\right\}\\&=\frac{\sqrt{2}}{8}W_8^{\frac{1}{2}}\left\{x_0+x_1W_8+x_2W_8^{\,2}+x_3W_8^{\,3}+x_3W_8^{\,4}+x_2W_8^{\,5}+x_1W_8^{\,6}+x_0W_8^{\,7}\right\}\end{aligned}\quad\cdots\cdots(17)$$

となる．

一方，8サンプルのディジタル信号，すなわち，

$$\{x_0,\ x_1,\ x_2,\ x_3,\ x_3,\ x_2,\ x_1,\ x_0\}$$

に対するDFT値$X_1^{(8)}$は，

$$X_1^{(8)}=\frac{1}{8}\left\{x_0+x_1W_8+x_2W_8^{\,2}+x_3W_8^{\,3}+x_3W_8^{\,4}+x_2W_8^{\,5}+x_1W_8^{\,6}+x_0W_8^{\,7}\right\}\quad\cdots\cdots(18)$$

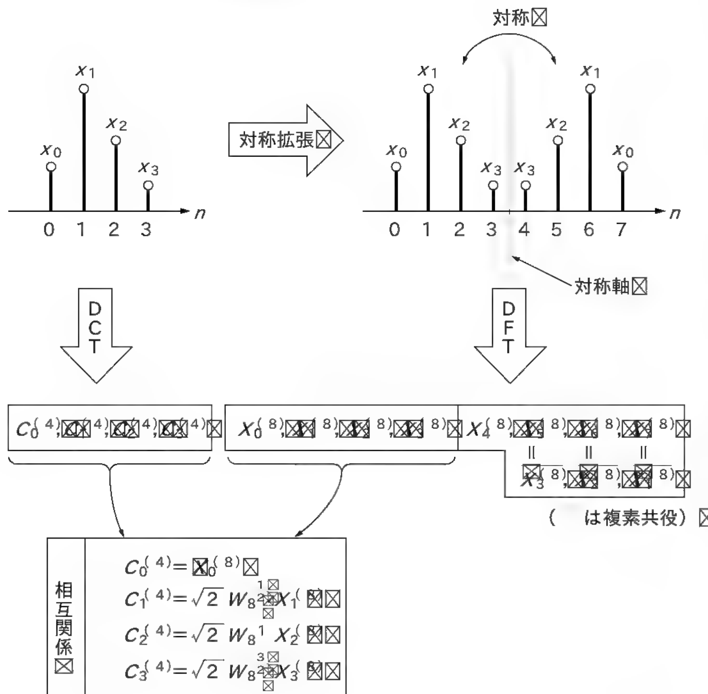
で与えられる[本連載の第4回「ディジタルフーリエ変換(DFT)の諸性質と一般化」，2001年10月号を参照]．つまり，式(18)のDFT値$X_1^{(8)}$は，4サンプルの信号$\{x_0,\ x_1,\ x_2,\ x_3\}$を対称に折り返した8サンプルの信号に対するDFT値を計算することに等価であることを意味している(図21.1)．

以上の考察から，式(17)と式(18)を比較することにより，DCT値$C_1^{(4)}$とDFT値$X_1^{(8)}$との間には，

$$C_1^{(4)}=\sqrt{2}W_8^{\frac{1}{2}}X_1^{(8)}\quad\cdots\cdots(19)$$



〔図21.1〕DCTとDFTの関係




〔図21.2〕DCTとDFTとの関係を導き出すプロセス($N=4$, $\ell \neq 0$)

$$C_\ell^{(4)} = \frac{1}{4}\left[x_0\left\{\sqrt{2}\cos\left(\frac{\ell\pi}{8}\right)\right\} + x_1\left\{\sqrt{2}\cos\left(\frac{3\ell\pi}{8}\right)\right\} + x_2\left\{\sqrt{2}\cos\left(\frac{5\ell\pi}{8}\right)\right\} + x_3\left\{\sqrt{2}\cos\left(\frac{7\ell\pi}{8}\right)\right\}\right]$$

↓ オイラーの公式(式(12))による

$$C_\ell^{(4)} = \frac{\sqrt{2}}{8}\left[x_0\left(W_8^{\frac{\ell}{2}} + W_8^{-\frac{\ell}{2}}\right) + x_1\left(W_8^{\frac{3\ell}{2}} + W_8^{-\frac{3\ell}{2}}\right) + x_2\left(W_8^{\frac{5\ell}{2}} + W_8^{-\frac{5\ell}{2}}\right) + x_3\left(W_8^{\frac{7\ell}{2}} + W_8^{-\frac{7\ell}{2}}\right)\right]$$

↓ $W_8^{-\frac{m\ell}{2}} = W_8^{\left(8-\frac{m}{2}\right)\ell}$

$$C_\ell^{(4)} = \frac{\sqrt{2}}{8}\left[x_0\left(W_8^{\frac{\ell}{2}} + W_8^{\frac{15\ell}{2}}\right) + x_1\left(W_8^{\frac{3\ell}{2}} + W_8^{\frac{13\ell}{2}}\right) + x_2\left(W_8^{\frac{5\ell}{2}} + W_8^{\frac{11\ell}{2}}\right) + x_3\left(W_8^{\frac{7\ell}{2}} + W_8^{\frac{9\ell}{2}}\right)\right]$$

↓ $W_8^{\frac{\ell}{2}}$をくくり出す

$$C_\ell^{(4)} = \frac{\sqrt{2}}{8} W_8^{\frac{\ell}{2}}\left[x_0\left(1 + W_8^{7\ell}\right) + x_1\left(W_8^{\ell} + W_8^{6\ell}\right) + x_2\left(W_8^{2\ell} + W_8^{5\ell}\right) + x_3\left(W_8^{3\ell} + W_8^{4\ell}\right)\right]$$

↓ 並び替えて整理する

$$C_\ell^{(4)} = \sqrt{2}\, W_8^{\frac{\ell}{2}} \underbrace{\frac{1}{8}\left(x_0 + x_1 W_8^{\ell} + x_2 W_8^{2\ell} + x_3 W_8^{3\ell} + x_3 W_8^{4\ell} + x_2 W_8^{5\ell} + x_1 W_8^{6\ell} + x_0 W_8^{7\ell}\right)}_{\text{DFT値 } X_\ell^{(8)}}$$

で表される関係が導かれる．すなわち，4サンプルの信号に対するDCT値が，8サンプルの信号に対するDFT値で計算できることになり，DFTの高速計算アルゴリズム“FFT”を適用できるのである．なお，FFTについては本連載の第10回(FFT計算アルゴリズムの一般化(2002年7月号))を参考にしてもらいたい．

**例題1**

式(3)に基づき，$\ell = 2$に対するDCT値$C_2^{(4)}$をDFT計算で求めよ．

**解答1**

図21.2のフローチャートに基づき，$W_8 = e^{-j\frac{2\pi}{8}}$を用いて，ていねいに計算するとよい．

$$C_2^{(4)} = \frac{\sqrt{2}}{8}\left\{x_0\cos\left(\frac{2\pi}{8}\right) + x_1\cos\left(\frac{6\pi}{8}\right) + x_2\cos\left(\frac{10\pi}{8}\right) + x_3\cos\left(\frac{14\pi}{8}\right)\right\} \quad \cdots\cdots(20)$$

$$\begin{aligned}C_2^{(4)} &= \frac{\sqrt{2}}{8}\left\{x_0\left(W_8^{1} + W_8^{-1}\right) + x_1\left(W_8^{3} + W_8^{-3}\right) + x_2\left(W_8^{5} + W_8^{-5}\right) + x_3\left(W_8^{7} + W_8^{-7}\right)\right\}\\ &= \frac{\sqrt{2}}{8}\left\{x_0\left(W_8 + W_8^{15}\right) + x_1\left(W_8^{3} + W_8^{13}\right) + x_2\left(W_8^{5} + W_8^{11}\right) + x_3\left(W_8^{7} + W_8^{9}\right)\right\}\end{aligned} \quad \cdots\cdots(21)$$

続けて，$W_8$をくくり出し，整理すると，

$$\begin{aligned}C_2^{(4)} &= \frac{\sqrt{2}}{8} W_8\left\{x_0\left(1 + W_8^{14}\right) + x_1\left(W_8^{2} + W_8^{12}\right) + x_2\left(W_8^{4} + W_8^{10}\right) + x_3\left(W_8^{6} + W_8^{8}\right)\right\}\\ &= \frac{\sqrt{2}}{8} W_8\left\{x_0 + x_1 W_8^{2} + x_2 W_8^{4} + x_3 W_8^{6} + x_3 W_8^{8} + x_2 W_8^{10} + x_1 W_8^{12} + x_0 W_8^{14}\right\}\end{aligned} \quad \cdots\cdots(22)$$

となる．

また，8サンプルのディジタル信号，すなわち，

$\{x_0, x_1, x_2, x_3, x_3, x_2, x_1, x_0\}$

に対するDFT値$X_2^{(8)}$は，

$$X_2^{(8)} = \frac{1}{8}\left\{x_0 + x_1 W_8^{2} + x_2 W_8^{4} + x_3 W_8^{6} + x_3 W_8^{8} + x_2 W_8^{10} + x_1 W_8^{12} + x_0 W_8^{14}\right\} \quad \cdots\cdots(23)$$

で与えられるので，式(22)と見比べることにより，

$$C_2^{(4)} = \sqrt{2}\, W_8 X_2^{(8)} \quad \cdots\cdots(24)$$

となる．

なお，$\ell = 0$, 3についても結果のみを示しておくので，各自で必ず検証しておいてほしい．

1) $\ell = 0$(直流)の場合

$$C_0^{(4)} = \frac{1}{4}\left\{x_0 + x_1 + x_2 + x_3\right\} \quad \cdots\cdots(25)$$

$$X_0^{(8)} = \frac{1}{8}\left\{x_0 + x_1 + x_2 + x_3 + x_3 + x_2 + x_1 + x_0\right\} = \frac{1}{4}\left\{x_0 + x_1 + x_2 + x_3\right\} \quad \cdots\cdots(26)$$

$$C_0^{(4)} = X_0^{(8)} \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (27)$$

2) $\ell = 3$の場合

$$C_3^{(4)} = \frac{\sqrt{2}}{8}\left\{x_0\cos\left(\frac{3\pi}{8}\right) + x_1\cos\left(\frac{9\pi}{8}\right) + x_2\cos\left(\frac{15\pi}{8}\right) + x_3\cos\left(\frac{21\pi}{8}\right)\right\}$$

$$= \frac{\sqrt{2}}{8}W_8^{\frac{3}{2}}\{x_0 + x_1W_8^3 + x_2W_8^6 + x_3W_8^9 + x_3W_8^{12} + x_2W_8^{15} + x_1W_8^{18} + x_0W_8^{21}\} \quad \cdot (28)$$

$$X_2^{(8)} = \frac{1}{8}\{x_0 + x_1W_8^3 + x_2W_8^6 + x_3W_8^9 + x_3W_8^{12} + x_2W_8^{15} + x_1W_8^{18} + x_0W_8^{21}\} \quad \cdots (29)$$

$$C_3^{(4)} = \sqrt{2}W_8^{\frac{3}{2}}X_3^{(8)} \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (30)$$

以上の結果に基づき，4サンプルのディジタル信号$\{x_0,\ x_1,\ x_2,\ x_3\}$に対するDCT値$\{C_\ell^{(4)}\}_{\ell=0}^{\ell=3}$のFFTによる高速計算処理の流れを図21.3に示しておく．ここで，図21.3の記号"□"は加算を，破線は乗算による計算処理を表す．

## スパース行列分解による高速計算アルゴリズム

FFTによるDCTの高速計算アルゴリズムでは，複素数演算で実行されるが，DCT計算は式(3)のように実数計算のみで表されている．たとえば，複素数計算では，加算と乗算は次のように計算される．

○ 複素加算

$$(a + jb) + (c + jd) = (a + c) + j(b + d) \quad \cdots\cdots\cdots (31)$$

実数加算＝2回

○ 複素乗算

$$(a + jb) \times (c + jd) = (ac - bd) + j(ad + bc) \quad \cdots\cdots (32)$$

実数加算＝2回

実数乗算＝4回

このように複素数演算は多くの実数演算を含むことになるので高速処理には不向きである．一方，DCT計算は式(3)からわかるように，実数演算のみで表されるので，複素数演算の形式ではなく，実数のみの演算をベースにした高速計算アルゴリズムが望ましいことに気づかされる．

そこで，DCT行列$\{\lambda_{\ell n}^{(N)}\}_{\ell,n=0}^{\ell,n=N-1}$において，"スパース(sparse)行列"とよばれるゼロ(0)要素が多い行列に分解し，複素数演算を用いずに高速計算するアルゴリズムが考えられている．ただ，行列分解法は唯一ではないので，さまざまなアルゴリズムが提案されている．ここでは，$N = 4$サンプルの場合を例に，チェン(Chen)らの提案した行列分解について説明する．

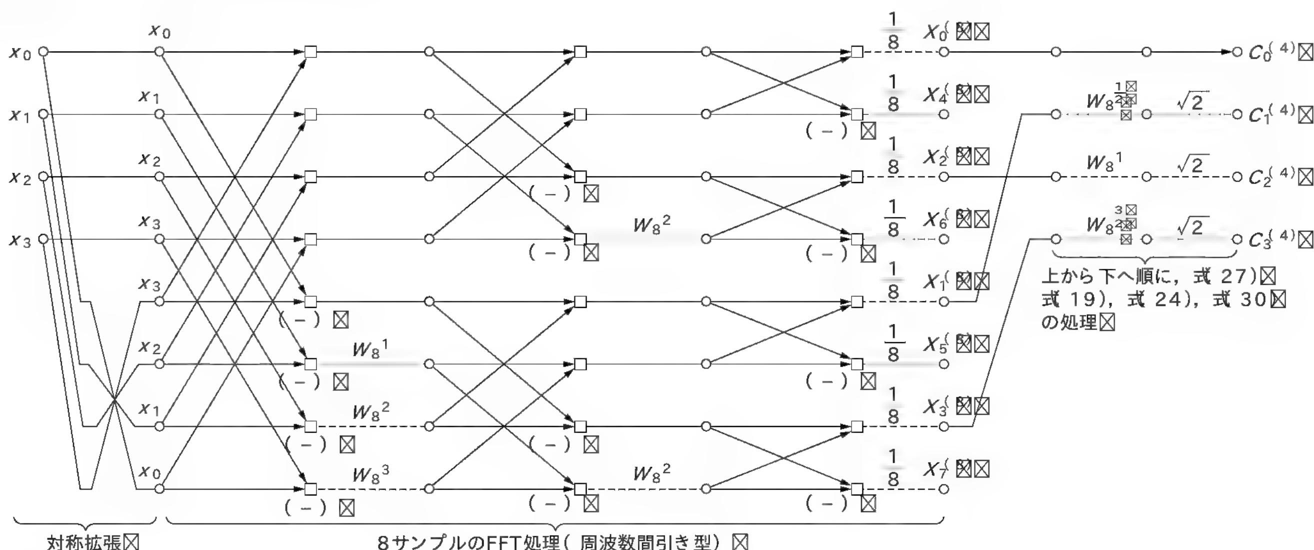
まず，式(3)～式(6)より，定数係数(1/4)を$\sqrt{2}/4$として式(5)を書き換えると，

$$\begin{pmatrix} C_0^{(4)} \\ C_1^{(4)} \\ C_2^{(4)} \\ C_3^{(4)} \end{pmatrix} = \frac{\sqrt{2}}{4}\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \alpha_8^1 & \alpha_8^3 & \alpha_8^5 & \alpha_8^7 \\ \alpha_8^2 & \alpha_8^6 & \alpha_8^{10} & \alpha_8^{14} \\ \alpha_8^3 & \alpha_8^9 & \alpha_8^{15} & \alpha_8^{21} \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \cdots (33)$$

$$\alpha_\ell^n = \cos\left(\frac{n\pi}{\ell}\right) \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots (34)$$

と表される．なお，式(4)のDCT行列の各要素$\{\lambda_{\ell n}^{(N)}\}_{\ell,n=0}^{\ell,n=N-1}$は式(34)の表記法をとれば，

〔図21.3〕FFTによるDCT計算処理の流れ($N = 4$)





$$\lambda_{\ell n}{}^{(N)} = \gamma_\ell \alpha_{2N}{}^{(2n+1)\ell} \quad \cdots\cdots(35)$$

となる．

次に，定数係数$\sqrt{2}/4$を除き，式(33)の偶数行と奇数行を集めて，上下に並べ替えてみよう．

$$C_\ell{}^{(4)} = \frac{\sqrt{2}}{4}\tilde{C}_\ell{}^{(4)} \quad \cdots\cdots(36)$$

$$\begin{pmatrix} \tilde{C}_0{}^{(4)} \\ \tilde{C}_2{}^{(4)} \\ \tilde{C}_1{}^{(4)} \\ \tilde{C}_3{}^{(4)} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \alpha_8{}^2 & \alpha_8{}^6 & \alpha_8{}^{10} & \alpha_8{}^{14} \\ \hline \alpha_8{}^1 & \alpha_8{}^3 & \alpha_8{}^5 & \alpha_8{}^7 \\ \alpha_8{}^3 & \alpha_8{}^9 & \alpha_8{}^{15} & \alpha_8{}^{21} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \cdots\cdots(37)$$

また，三角関数の性質を利用し，式(34)の簡略表記を適用すれば，

$$\begin{cases} \alpha_8{}^2 = \cos\left(\frac{2\pi}{8}\right) = \cos\left(\frac{\pi}{4}\right) = \alpha_4{}^1 \\ \alpha_8{}^6 = \cos\left(\frac{6\pi}{8}\right) = \cos\left(\frac{3\pi}{4}\right) = \cos\left(\pi - \frac{\pi}{4}\right) = -\cos\left(\frac{\pi}{4}\right) = -\alpha_4{}^1 \\ \alpha_8{}^{10} = \cos\left(\frac{10\pi}{8}\right) = \cos\left(\frac{5\pi}{4}\right) = \cos\left(\pi + \frac{\pi}{4}\right) = -\cos\left(\frac{\pi}{4}\right) = -\alpha_4{}^1 \\ \alpha_8{}^{14} = \cos\left(\frac{14\pi}{8}\right) = \cos\left(\frac{7\pi}{4}\right) = \cos\left(2\pi - \frac{\pi}{4}\right) = \cos\left(\frac{\pi}{4}\right) = \alpha_4{}^1 \\ \alpha_8{}^5 = \cos\left(\frac{5\pi}{8}\right) = \cos\left(\pi - \frac{5\pi}{8}\right) = -\cos\left(\frac{3\pi}{8}\right) = -\alpha_8{}^3 \\ \alpha_8{}^7 = \cos\left(\frac{7\pi}{8}\right) = \cos\left(\pi - \frac{7\pi}{8}\right) = -\cos\left(\frac{\pi}{8}\right) = -\alpha_8{}^1 \\ \alpha_8{}^9 = \cos\left(\frac{9\pi}{8}\right) = \cos\left(\pi + \frac{\pi}{8}\right) = -\cos\left(\frac{\pi}{8}\right) = -\alpha_8{}^1 \\ \alpha_8{}^{15} = \cos\left(\frac{15\pi}{8}\right) = \cos\left(2\pi - \frac{\pi}{8}\right) = \cos\left(\frac{\pi}{8}\right) = \alpha_8{}^1 \\ \alpha_8{}^{21} = \cos\left(\frac{21\pi}{8}\right) = \cos\left(2\pi + \frac{5\pi}{8}\right) = \cos\left(\frac{5\pi}{8}\right) = \alpha_8{}^5 = -\alpha_8{}^3 \end{cases}$$

$$\cdots\cdots(38)$$

となる関係が成立するので，式(33)は次のように書き換えることができる．

$$\begin{pmatrix} \tilde{C}_0{}^{(4)} \\ \tilde{C}_2{}^{(4)} \\ \tilde{C}_1{}^{(4)} \\ \tilde{C}_3{}^{(4)} \end{pmatrix} = \left(\begin{array}{cc|cc} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \alpha_4{}^1 & -\alpha_4{}^1 & -\alpha_4{}^1 & \alpha_4{}^1 \\ \hline \alpha_8{}^1 & \alpha_8{}^3 & -\alpha_8{}^3 & -\alpha_8{}^1 \\ \alpha_8{}^3 & -\alpha_8{}^1 & \alpha_8{}^1 & -\alpha_8{}^3 \end{array}\right) \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \cdots\cdots(39)$$

ところで，式(34)より，

$$\frac{1}{\sqrt{2}} = \cos\left(\frac{\pi}{4}\right) = \alpha_4{}^1 \quad \cdots\cdots(40)$$

であり，さらに，

$$\beta_\ell{}^n = \sin\left(\frac{n\pi}{\ell}\right) \quad \cdots\cdots(41)$$

と表すことにすると，

$$\alpha_8{}^3 = \cos\left(\frac{3\pi}{8}\right) = \cos\left(\frac{\pi}{2} - \frac{\pi}{8}\right) = \sin\left(\frac{\pi}{8}\right) = \beta_8{}^1 \quad \cdots(42)$$

$$\alpha_8{}^1 = \cos\left(\frac{\pi}{8}\right) = \cos\left(\frac{\pi}{2} - \frac{3\pi}{8}\right) = \sin\left(\frac{3\pi}{8}\right) = \beta_8{}^3 \quad \cdots(43)$$

となる関係を用いて，式(39)は次のように書き換えられる．

$$\begin{pmatrix} \tilde{C}_0{}^{(4)} \\ \tilde{C}_2{}^{(4)} \\ \tilde{C}_1{}^{(4)} \\ \tilde{C}_3{}^{(4)} \end{pmatrix} = \left(\begin{array}{cc|cc} \alpha_4{}^1 & \alpha_4{}^1 & \alpha_4{}^1 & \alpha_4{}^1 \\ \alpha_4{}^1 & -\alpha_4{}^1 & -\alpha_4{}^1 & \alpha_4{}^1 \\ \hline \alpha_8{}^1 & \beta_8{}^1 & -\beta_8{}^1 & -\alpha_8{}^1 \\ \alpha_8{}^3 & -\beta_8{}^3 & \beta_8{}^3 & -\alpha_8{}^3 \end{array}\right) \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \cdots(44)$$

式(33)において，定数係数$\sqrt{2}/4$を除いたDCT行列を$\boldsymbol{A}_4$と表すことにし，

$$\boldsymbol{A}_2 = \begin{bmatrix} \alpha_4{}^1 & \alpha_4{}^1 \\ \alpha_4{}^1 & -\alpha_4{}^1 \end{bmatrix} \quad \cdots\cdots(45)$$

$$\boldsymbol{R}_2 = \begin{bmatrix} \alpha_8{}^1 & \beta_8{}^1 \\ \alpha_8{}^3 & -\beta_8{}^3 \end{bmatrix} \quad \cdots\cdots(46)$$

$$\boldsymbol{I}_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{（単位行列）} \quad \cdots\cdots(47)$$

$$\boldsymbol{J}_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \cdots\cdots(48)$$

と定義すると，式(44)のDCT行列は次のように表される．

$$\boldsymbol{B}_4\boldsymbol{A}_4 = \begin{bmatrix} \boldsymbol{A}_2 & \boldsymbol{A}_2\boldsymbol{J}_2 \\ \boldsymbol{R}_2 & -\boldsymbol{R}_2\boldsymbol{J}_2 \end{bmatrix} \quad \cdots\cdots(49)$$

ただし，

$$\boldsymbol{B}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \cdots\cdots(50)$$

であり，置換行列とよばれる．なお，各行列の下付き数字は行列の大きさを表し，数字の4は4行4列，2は2行2列を意味する．

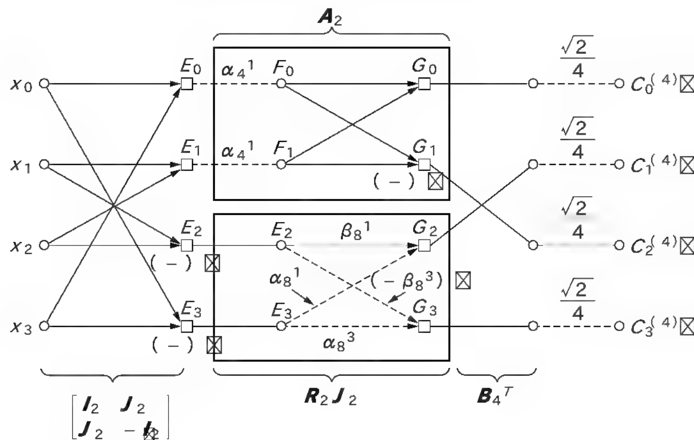
ちなみに，式(49)が式(44)のDCT行列に一致することは，以下の計算により容易に確かめられる．

$$\boldsymbol{A}_2\boldsymbol{J}_2 = \begin{bmatrix} \alpha_4{}^1 & \alpha_4{}^1 \\ \alpha_4{}^1 & -\alpha_4{}^1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \alpha_4{}^1 & \alpha_4{}^1 \\ -\alpha_4{}^1 & \alpha_4{}^1 \end{bmatrix} \quad \cdots\cdots(51)$$

$$-\boldsymbol{R}_2\boldsymbol{J}_2 = -\begin{bmatrix} \alpha_8{}^1 & \beta_8{}^1 \\ \alpha_8{}^3 & -\beta_8{}^3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} -\beta_8{}^1 & -\alpha_8{}^1 \\ \beta_8{}^3 & -\alpha_8{}^3 \end{bmatrix} \quad \cdots(52)$$

続いて，$\boldsymbol{B}_4$の転置行列$\boldsymbol{B}_4{}^{\mathrm{T}}$，すなわち，

〔図21.4〕スパース行列分解によるDCT計算処理の流れ（$N = 4$）




〔表21.1〕演算量の比較

| サンプル数 $N$ | 直接計算法 | 行列分解法 |
|---|---|---|
| 4（$= 2^2$） | 16 | 6 |
| 8（$= 2^3$） | 64 | 16 |
| 16（$= 2^4$） | 256 | 44 |
| 32（$= 2^5$） | 1024 | 116 |
| 64（$= 2^6$） | 4096 | 292 |
| 128（$= 2^7$） | 16384 | 708 |

（a）実数乗算

| サンプル数 $N$ | 直接計算法 | 行列分解法 |
|---|---|---|
| 4（$= 2^2$） | 12 | 8 |
| 8（$= 2^3$） | 56 | 26 |
| 16（$= 2^4$） | 240 | 74 |
| 32（$= 2^5$） | 992 | 194 |
| 64（$= 2^6$） | 4032 | 482 |
| 128（$= 2^7$） | 16256 | 1154 |

（b）実数加算

$$B_4^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \cdots\cdots (53)$$

を用いて，式（49）より，

$$A_4 = B_4^{\mathrm{T}} \begin{bmatrix} A_2 & A_2 J_2 \\ R_2 & -R_2 J_2 \end{bmatrix} \quad \cdots\cdots (54)$$

となる関係が得られる．また，$I_2$と$J_2$について，

$$J_2 J_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I_2 \quad \cdots\cdots (55)$$

$$J_2 I_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = J_2 \quad \cdots\cdots (56)$$

で表される性質があること，さらに式（47）の単位行列$I_2$に対して，

$$\begin{cases} A_2 = A_2 I_2 \\ R_2 = R_2 I_2 \\ -R_2 J_2 = R_2 J_2 (-I_2) \end{cases} \quad \cdots\cdots (57)$$

となる関係が成り立つことを考慮し，式（54）の［　］を次のように書き換える．

$$\begin{bmatrix} A_2 & A_2 J_2 \\ R_2 & -R_2 J_2 \end{bmatrix} = \begin{bmatrix} A_2 I_2 & A_2 J_2 \\ R_2 I_2 & -R_2 J_2 I_2 \end{bmatrix} = \begin{bmatrix} A_2 I_2 & A_2 J_2 \\ R_2 J_2 J_2 & -R_2 J_2 I_2 \end{bmatrix} = \begin{bmatrix} A_2 I_2 & A_2 J_2 \\ R_2 J_2 J_2 & R_2 J_2 (-I_2) \end{bmatrix} \quad \cdots (58)$$

よって，式（58）は，

$$\begin{bmatrix} A_2 I_2 & A_2 J_2 \\ R_2 J_2 J_2 & R_2 J_2 (-I_2) \end{bmatrix} = \begin{bmatrix} A_2 & O_2 \\ O_2 & R_2 J_2 \end{bmatrix} \begin{bmatrix} I_2 & J_2 \\ J_2 & -I_2 \end{bmatrix} \quad \cdots\cdots (59)$$

$$\text{ただし，} R_2 J_2 = \begin{bmatrix} \beta_8^1 & \alpha_8^1 \\ -\beta_8^3 & \alpha_8^3 \end{bmatrix}$$

と表される．ここで，

$$O_2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \cdots\cdots (60)$$

であり，零行列と呼ばれる．この零行列をなるべく多くするようにDCT行列を分解する（スパース化する）ことが，高速化に寄与することになる．

最終的には，DCT行列$A_4$は，

$$A_4 = B_4^{\mathrm{T}} \begin{bmatrix} A_2 & O_2 \\ O_2 & R_2 J_2 \end{bmatrix} \begin{bmatrix} I_2 & J_2 \\ J_2 & -I_2 \end{bmatrix} \quad \cdots\cdots (61)$$

と分解される．そこで，式（61）に基づき，スパース行列分解による高速計算処理の流れを図21.4に示しておくので，各自でDCT値が得られることを確認してもらいたい．なお，サンプル数$N$が2のべき乗の場合，式（61）の分解は繰り返し行うことができ，高速処理が実現されることが知られている．

このとき図21.4より，演算量は，

$$\begin{cases} \text{実数乗算} = 6\text{回} \\ \text{実数加算} = 8\text{回} \end{cases}$$

となっている．一方，式（33）を直接計算する場合は，式（7）に$N = 4$を代入して，

$$\begin{cases} \text{実数乗算} = 16\text{回} \\ \text{実数加算} = 12\text{回} \end{cases}$$



となり，行列分解することで演算量を減らせることが実感できる．一般に$N$サンプルに対して，行列分解による演算量は，

$$\begin{cases} 実数乗算 = N\ell\mathrm{og}_2(N) - \dfrac{3N}{2} + 4 \\ 実数加算 = \dfrac{3N}{2}\{\ell\mathrm{og}_2(N) - 1\} + 2 \quad ; \; N \geqq 4 \end{cases} \quad \cdots\cdots\cdots(62)$$

であり，直接計算〔式(7)〕と行列分解とを比較した演算回数を表21.1に示す．

**例題2**

図21.4において，出力$\{C_\ell^{(4)}\}_{\ell=0}^{\ell=3}$が式(3)のDCT値に相当することを確認せよ．

**解答2**

式(33)～式(61)を参考に，図21.4の計算処理をまとめておく．その際，とくに式(34)，式(38)，式(40)，式(41)～式(43)に注意し，手間をおしまず，ていねいに確認してもらいたい．なお，DCTの逆変換(IDCT)の高速処理は，図21.4の信号処理の流れを示す矢印の向きを右から左へ逆向きにすることで実現できる．その際，定数係数$\sqrt{2}/4$は$\sqrt{2}$になる．

● 第1段目

$E_0 = x_0 + x_3, \quad E_1 = x_1 + x_2$

$E_2 = x_1 - x_2, \quad E_3 = x_0 - x_3$

● 第2段目

$F_0 = \alpha_4^1 E_0, \quad F_1 = \alpha_4^1 E_1$

● 第3段目

$$\begin{aligned} G_0 &= F_0 + F_1 \\ &= \alpha_4^1(x_0 + x_3) + \alpha_4^1(x_1 + x_2) \\ &= \frac{1}{\sqrt{2}}x_0 + \frac{1}{\sqrt{2}}x_1 + \frac{1}{\sqrt{2}}x_2 + \frac{1}{\sqrt{2}}x_3 \end{aligned}$$

$$\begin{aligned} G_1 &= F_0 - F_1 \\ &= \alpha_4^1(x_0 + x_3) - \alpha_4^1(x_1 + x_2) \\ &= \alpha_4^1 x_0 - \alpha_4^1 x_1 - \alpha_4^1 x_2 + \alpha_4^1 x_3 \\ &= \alpha_8^2 x_0 - \alpha_8^2 x_1 - \alpha_8^2 x_2 + \alpha_8^2 x_3 \\ &= \alpha_8^2 x_0 + \alpha_8^6 x_1 + \alpha_8^{10} x_2 + \alpha_8^{14} x_3 \end{aligned}$$

$$\begin{aligned} G_2 &= \beta_8^1 E_2 + \alpha_8^1 E_3 \\ &= \beta_8^1(x_1 - x_2) + \alpha_8^1(x_0 - x_3) \\ &= \alpha_8^1 x_0 + \beta_8^1 x_1 - \beta_8^1 x_2 - \alpha_8^1 x_3 \\ &= \alpha_8^1 x_0 + \alpha_8^3 x_1 - \alpha_8^3 x_2 - \alpha_8^1 x_3 \\ &= \alpha_8^1 x_0 + \alpha_8^3 x_1 + \alpha_8^5 x_2 + \alpha_8^7 x_3 \end{aligned}$$

$$\begin{aligned} G_3 &= -\beta_8^3 E_2 + \alpha_8^3 E_3 \\ &= -\beta_8^3(x_1 - x_2) + \alpha_8^3(x_0 - x_3) \\ &= \alpha_8^3 x_0 - \beta_8^3 x_1 + \beta_8^3 x_2 - \alpha_8^3 x_3 \\ &= \alpha_8^3 x_0 - \alpha_8^1 x_1 + \alpha_8^1 x_2 - \alpha_8^3 x_3 \\ &= \alpha_8^3 x_0 + \alpha_8^9 x_1 + \alpha_8^{15} x_2 + \alpha_8^{21} x_3 \end{aligned}$$

● 第4段目

$$C_0^{(4)} = \frac{\sqrt{2}}{4}G_0 = \frac{1}{4}\{x_0 + x_1 + x_0 + x_1\}$$

$$\begin{aligned} C_1^{(4)} &= \frac{\sqrt{2}}{4}G_2 \\ &= \frac{\sqrt{2}}{4}\left\{x_0\cos\left(\frac{\pi}{8}\right) + x_1\cos\left(\frac{3\pi}{8}\right) \right. \\ &\qquad \left. + x_2\cos\left(\frac{5\pi}{8}\right) + x_3\cos\left(\frac{7\pi}{8}\right)\right\} \end{aligned}$$

$$\begin{aligned} C_2^{(4)} &= \frac{\sqrt{2}}{4}G_1 \\ &= \frac{\sqrt{2}}{4}\left\{x_0\cos\left(\frac{2\pi}{8}\right) + x_1\cos\left(\frac{6\pi}{8}\right) \right. \\ &\qquad \left. + x_2\cos\left(\frac{10\pi}{8}\right) + x_3\cos\left(\frac{14\pi}{8}\right)\right\} \end{aligned}$$

$$\begin{aligned} C_3^{(4)} &= \frac{\sqrt{2}}{4}G_3 \\ &= \frac{\sqrt{2}}{4}\left\{x_0\cos\left(\frac{3\pi}{8}\right) + x_1\cos\left(\frac{9\pi}{8}\right) \right. \\ &\qquad \left. + x_2\cos\left(\frac{15\pi}{8}\right) + x_3\cos\left(\frac{21\pi}{8}\right)\right\} \end{aligned}$$

次回は，“実務に直結して応用できるDCTアプリケーション”として，画像データ処理を中心にわかりやすく解説する予定である．お楽しみに．

みたに・まさあき　東京電機大学 工学部情報通信工学科

# CQ RISC評価キット SH-4PCI with Linux 活用研究5

## Linux上から各種USB機器を使う

酒匂 信尋

CQ RISC評価キット/SH-4PCI with LinuxのSH-4ボードには12Mbpsおよび1.5Mbpsまでに対応したUSBホストコントローラが搭載されている．Linuxもカーネル2.4なので，基本的にはLinuxからUSB機器が使えるはずである．しかし本キットの出荷状態ではドライバなどが用意されていないためUSBは非対応となっている．そこでドライバなどに修正を加え，Linux上からUSB機器を使えるように設定してみる．（編集部）

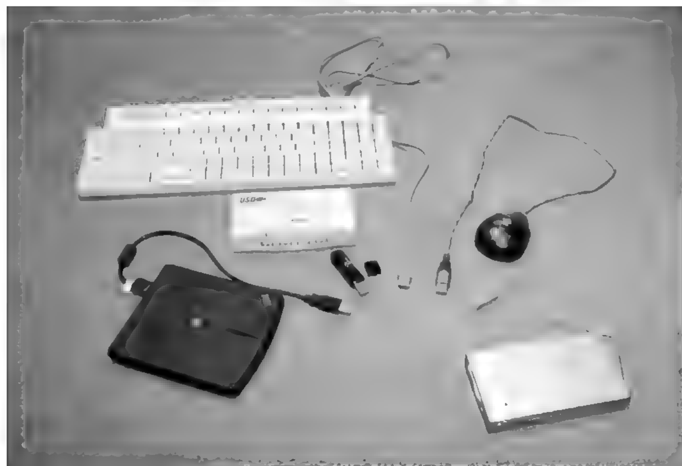
### はじめに

ひと昔前までは，CD-ROMドライブやイメージスキャナなどの周辺機器を拡張して使用する際のインターフェースとしてSCSIが使われていました．しかし最近ではUSBを使うケースも増えています．マウスやキーボードはもちろん，EthernetやHDDなど，接続できない機器はないと思えるほどです．組み込み機器においては，実際に使用する機能だけを実装した最小限の構成としたいところですが，機器の保守や将来的な機能拡張などを考えて，USBも使えるようにしておくことは，かなり有効ではないかと思います．

Linuxはカーネル2.4からUSBをサポートしています．またCQ RISC評価キット/SH-4PCI with Linuxの評価ボードにはUSBのホストコントローラも内蔵されているので，カーネルの設定を正しく行えば，USB機器を接続できるはずです．

そこで今回は，この評価キットに各種USB機器を接続してみましょう．写真1に，今回の接続テストで使用したUSB機器を示します．

〔写真1〕テストで使用したUSB機器
（HDD，マウス，キーボード，USBフラッシュメモリスティック，ハブ，CD-ROMドライブ）



## 1 PCIバスアクセス制御部の修正

● PCIバスマスタアクセス

PCIデバイスには，PCIバス上のデバイスが制御権を取得して，ホストCPUのメインメモリに対して直接データを読み書きしてくるバスマスタデバイスが存在します．ここで，そのメインメモリの領域がCPUのキャッシュにヒットしている場合，CPUが書き込んだデータはキャッシュ上だけに存在し，そのアドレスに対応する実メモリ上は古いデータのままとなります．もし，この状態のままでPCIバスマスタデバイスがそのメモリを読み出すと，古いデータを読み出してしまいます．

そこでx86系プロセッサでは，CPUがキャッシュしている空間に対してPCIバスマスタがアクセスすると，自動的にキャッシュをフラッシュしてからバスマスタへアクセスを行います．

ところがSHを含むRISC系プロセッサでは，x86のキャッシュのようなバスマスタデバイスに対して自動的にキャッシュフラッシュが発生するようなキャッシュ機構をもつものは少なく，このキャッシュフラッシュの制御を，ソフトウェアで解決しなくてはなりません．

● SH用LinuxとPCIバスマスタドライバ

カーネル2.4の初期版のSH用のLinuxでは，PCIバスへの対応が不十分でした．この評価キットに添付されているバージョンも2.4.5で，PCIへの対応が十分でありません．そのため，評価ボードオンボードのEthernetのドライバのソースでは，SH用に専用の修正が行われています．しかしこの手法では，USBでもそれぞれの機器ごとにドライバを修正しなくてはなりません．

そこで，それぞれのドライバごとに修正を加えるのではなく，SH-4のキャッシュとPCIバスの初期化制御のドライバ部分を修正することで，それぞれのPCIバスマスタ用ドライバは修正しなくても良いようにしてみます．

● キャッシュコントロール部

キャッシュコントロールの標準関数として，次の三つが`io.h`のヘッダファイルに定義されています．

（1）`dma_cache_wback_inv(_start, _size)`

実メモリにキャッシュの内容を反映し，キャッシュの内容を

〔リスト 1〕io.hの修正

```
--- linux/include/asm/io.h.org   Wed Jun  4 18:00:03 2003
+++ linux/include/asm/io.h  Wed Jun  4 18:03:38 2003
@@ -477,11 +477,11 @@
  */

 #define dma_cache_wback_inv(_start,_size) ¥
-    cache_flush_area((unsigned long)(_start),
                      ((unsigned long)(_start)+(_size)))
+    __flush_purge_region(_start,_size)
 #define dma_cache_inv(_start,_size) ¥
-    cache_purge_area((unsigned long)(_start),
                      ((unsigned long)(_start)+(_size)))
+    __flush_invalidate_inv(_start,_size)
 #define dma_cache_wback(_start,_size) ¥
-    cache_wback_area((unsigned long)(_start),
                      ((unsigned long)(_start)+(_size)))
+    __flush_wback_region(_start,_size)

 #endif /* __KERNEL__ */
 #endif /* __ASM_SH_IO_H */
```

無効にする．

(2) **dma_cache_inv(_start, _size)**

キャッシュの内容を無効にする．

(3) **dma_cache_wback(_start, _size)**

キャッシュの内容を有効にしたまま，実メモリにキャッシュの内容を反映する．

これら関数は，x86などのキャッシュ制御をハードウェアで自動的に処理してくれるCPU用のカーネルではノンオペレーションとなっていますが，MIPSやPA-RISC，ARM，そしてSHの場合は，ソフトウェアでキャッシュと実メモリのつじつまを合わせなければなりません．つまりこれらの関数の中身を適切に記述して実装する必要があります．

もともとこれらの関数は，CPUに依存しないように作成するデバイスドライバ内で使用するように定義されたもののようですが，実際はあまり使用されていないようです(とくにx86上では)．しかし，デバイスドライバで直接呼び出すことはなくても，PCIデバイスとのメモリの初期化などで必要になります．

修正の必要があるファイルは，次の二つです．

linux/include/asm/io.h

linux/arch/sh/mm/cache.c

修正部分をリスト1とリスト2に示します．

● PCIバスの初期化など

PCIバスのコントロール用の関数はpci.hで定義されています．主な関数を次に示します．この関数の中でキャッシュコントロールを吸収できれば，ほとんどのx86用のドライバを，修正なしでSH上で動かすことができます．

(1) **void *pci_alloc_consistent()**

デバイスとの共有メモリを確保して，CPU側，デバイス側から見たアドレスを返します．

(2) **void pci_free_consistent()**

共有メモリを返却します．

〔リスト 2〕cache.cの修正

```
--- linux/arch/sh/mm/cache.c.org Wed Jun  4 17:18:49 2003
+++ linux/arch/sh/mm/cache.c     Wed Jun  4 18:29:56 2003
@@ -203,6 +203,67 @@
 }

 /*
+ * Write back the dirty D-caches, but not invalidate them.
+ *
+ * START: Virtual Address (U0, P1, or P3)
+ * SIZE: Size of the region.
+ */
+void __flush_wback_region(void *start, int size)
+{
+   unsigned long v;
+   unsigned long begin, end;
+
+   begin = (unsigned long)start & ~(L1_CACHE_BYTES-1);
+   end = ((unsigned long)start + size + L1_CACHE_BYTES-1)
+         & ~(L1_CACHE_BYTES-1);
+   for (v = begin; v < end; v+=L1_CACHE_BYTES) {
+         asm volatile("ocbwb %0"
+                      : /* no output */
+                      : "m" (__m(v)));
+   }
+}
+
+/*
+ * Write back the dirty D-caches and invalidate them.
+ *
+ * START: Virtual Address (U0, P1, or P3)
+ * SIZE: Size of the region.
+ */
+void __flush_purge_region(void *start, int size)
+{
+   unsigned long v;
+   unsigned long begin, end;
+
+   begin = (unsigned long)start & ~(L1_CACHE_BYTES-1);
+   end = ((unsigned long)start + size + L1_CACHE_BYTES-1)
+         & ~(L1_CACHE_BYTES-1);
+   for (v = begin; v < end; v+=L1_CACHE_BYTES) {
+         asm volatile("ocbp %0"
+                      : /* no output */
+                      : "m" (__m(v)));
+   }
+}
+
+
+/*
+ * No write back please, just invalidate
+ */
+void __flush_invalidate_region(void *start, int size)
+{
+   unsigned long v;
+   unsigned long begin, end;
+
+   begin = (unsigned long)start & ~(L1_CACHE_BYTES-1);
+   end = ((unsigned long)start + size + L1_CACHE_BYTES-1)
+         & ~(L1_CACHE_BYTES-1);
+   for (v = begin; v < end; v+=L1_CACHE_BYTES) {
+         asm volatile("ocbi %0"
+                      : /* no output */
+                      : "m" (__m(v)));
+   }
+}
+
+/*
   * Invalidate I-caches.
   *
   * START, END: Virtual Address
```

(3) **dma_addr_t pci_map_single()**

バッファをフラッシュして，デバイス側から見たアドレスを返します．

(4) **void pci_unmap_single()**

pci_map_single()でリソースを確保していれば，それを

〔リスト 3〕**pci.h**の修正

```
--- linux/include/asm/pci.h.org  Wed Jun  4 14:41:07 2003
+++ linux/include/asm/pci.h Wed Jun  4 18:36:52 2003
@@ -100,10 +100,10 @@
                       size_t size,int directoin)
 {

+        dma_cache_wback_inv(ptr, size);
 #if defined(CONFIG_SH_KZP01)
    return pci_virt_to_bus(ptr);
 #else
-        flush_cache_all();
    return virt_to_bus(ptr);
 #endif
 }
@@ -139,7 +139,9 @@
 static inline int pci_map_sg(struct pci_dev *hwdev,
            struct scatterlist *sg, int nents,int direction)
 {
-        flush_cache_all();
+   int i;
+   for(i=0; i<nents; i++)
+            dma_cache_wback_inv(sg[i].address,
                                                sg[i].length);
    return nents;
 }

@@ -166,7 +168,11 @@
                         dma_addr_t dma_handle,
                         size_t size,int direction)
 {
-   /* Nothing to do */
+#if defined(CONFIG_SH_KZP01)
+   dma_cache_wback_inv(pci_bus_to_virt(dma_handle),size);
+#else
+   dma_cache_wback_inv(bus_to_virt(dma_handle),size);
+#endif
 }

 /* Make physical memory consistent for a set of streaming
@@ -175,11 +181,14 @@
  * The same as pci_dma_sync_single but for
                                        a scatter-gather list,
  * same rules and usage.
  */
-static inline void pci_dma_sync_sg(struct pci_dev *hwdev,
+static inline int pci_dma_sync_sg(struct pci_dev *hwdev,
                      struct scatterlist *sg,
                      int nelems,int direction)
 {
-   /* Nothing to do */
+   int i;
+   for(i=0; i<nelems; i++)
+      dma_cache_wback_inv(sg[i].address,sg[i].length);
+   return nelems;
 }


@@ -202,7 +211,11 @@
  * returns, or alternatively stop on the first
                                        sg_dma_len(sg) which
  * is 0.
  */
+#if defined(CONFIG_SH_KZP01)
+#define sg_dma_address(sg) (pci_virt_to_bus((sg)->address))
+#else
 #define sg_dma_address(sg) (virt_to_bus((sg)->address))
+#endif
 #define sg_dma_len(sg)          ((sg)->length)

 #endif /* __KERNEL__ */
```

〔リスト 4〕**pci_dma.c**の修正

```
--- linux/arch/sh/kernel/pci-dma.c.org  Wed Jun  4 14:37:36 2003
+++ linux/arch/sh/kernel/pci-dma.c  Wed Jun  4 14:38:14 2003
@@ -43,23 +43,14 @@
         *dma_handle = pci_virt_to_bus(ret);
    }
    /* We must flush the cache before we pass it on to
                                             the device */
-#if defined(CONFIG_SH_KZP01) // For debug 011211
-   return  ret;
-#else
    flush_cache_all();
    return  P2SEGADDR(ret);
-#endif
 }

 void pci_free_consistent(struct pci_dev *hwdev, size_t size,
              void *vaddr, dma_addr_t dma_handle)
 {
-
-#if defined(CONFIG_SH_KZP01) // For debug 011211
-        free_pages((unsigned long)vaddr, get_order(size));
-#else
        unsigned long p1addr=P1SEGADDR((unsigned long)vaddr);

    free_pages(p1addr, get_order(size));
-#endif
 }
```

〔リスト 5〕USBを有効に設定する

```
--- linux.org/arch/sh/config.in  Tue Apr  2 14:55:37 2002
+++ linux/arch/sh/config.in Thu May 29 21:04:10 2003
@@ -317,6 +317,8 @@
 fi
 endmenu

+source drivers/usb/Config.in
+
 mainmenu_option next_comment
 comment 'Kernel hacking'
```

返却します．

（5）**int pci_map_sg()**

HDDアクセスのバッファ管理に使用される，スキャッタリストのバッファをフラッシュします．

（6）***void pci_unmap_sg()**

pci_map_sg()でリソースを確保していれば，それを返却します．

（7）**void pci_dma_sync_single()**

バッファをフラッシュします．

（8）**void pci_dma_sync_sg()**

スキャッタリストで管理されているバッファをフラッシュします．

これらの修正ファイルは，次の二つです．

linux/include/asm/pci.h
linux/arch/sh/pci_dma.c

修正内容をリスト 3とリスト 4に示します．

● 修正後の動作確認

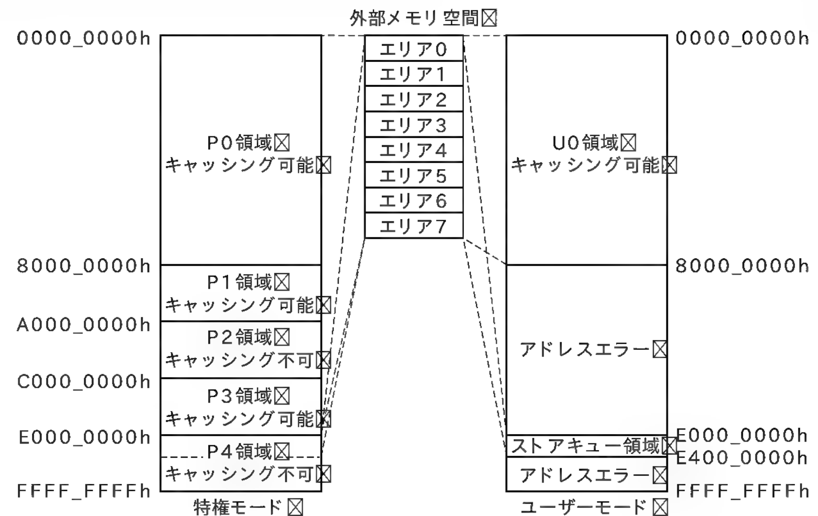
ソースの修正が終了したら，評価ボードオンボードのEthernetコントローラのドライバlinux/drivers/net/pcnet32.cを，パッチの入っていないオリジナルのソースと入れ替え，カーネルの再構築を行います．

こうして起動したシステムで，ネットワークが正しく動作することを確認してください．これが正常に動作すれば修正成功で，USB関連のドライバも修正なしで動くことが期待できます．

Column

## SH-4のキャッシュとPCIバスマスタコントロール

〔図A〕SH-4のメモリ空間

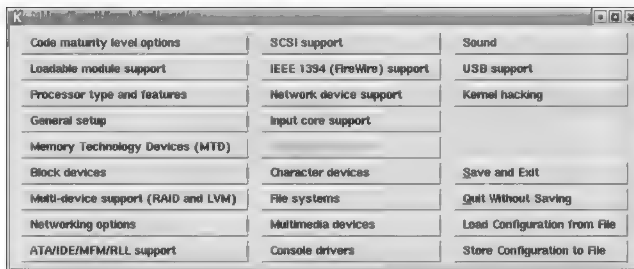



SH-4の場合，キャッシュコントロールはソフトウェアで制御すると説明しましたが，実際にリストを見ると，たいして特別な処理を行っていないように思われたかもしれません．

SH-4はアドレス空間としては32ビットですが，CPUの外部アドレスは29ビットとなっています．上位の3ビットはP0～P4の五つの領域の区別のために使われています（図A）．この領域についての詳細はハードウェアマニュアルを参照してください．

カーネルの動作空間はP1と呼ばれるキャッシングが可能な空間となっています．この空間でPCIバスマスタデバイスとメモリを共有すると，CPU側はキャッシュの内容を参照し，PCIバスマスタデバイスは実メモリを参照することになり，キャッシュの内容をつねにフラッシュしていないと，CPUがメモリに書いたつもりが実メモリには書かれておらず，不具合が発生してしまいます．これを防ぐ手段として，共有メモリはP2と呼ぶキャッシング不可の空間を使うようにしています．

〔図1〕カーネルにUSBを組み込む




〔図2〕USBホスト/マスストレージ/HIDの選択




## 2 USBドライバの組み込み

● USBデバイス選択メニューの表示

本評価キットの開発環境のままでは，カーネルの設定でUSBデバイス選択のメニューが出てきません．そこでメニューにUSBの項目が出てくるように，`linux/arch/config.in`を修正します．修正内容をリスト5に示します．

修正後に，

```
make xconfig
```

を実行すると，図1のように表示されます．

● USBデバイスの選択

ホストコントローラはUHCIとOHCIから選択可能です．本評価ボードにはOHCI仕様のホストコントローラが実装されているので，ここではOHCIを選択します．

USBターゲットデバイスとしては，USB接続のHDDやCD-ROMドライブを接続するなら“USB Mass Storage support”（マスストレージ）を，キーボードやマウスを接続するなら“Human Interface Devices”（HID）を選択します．選択後のメニュー画面を図2に示します．

〔図3〕SCSI CD-ROMの選択




〔リスト6〕SCSI接続のCD-ROMドライブのデバイス名

```
hub.c: port 1 connection change
hub.c: port 1, portstatus 101, change 1, 12 Mb/s
hub.c: port 1, portstatus 103, change 10, 12 Mb/s
hub.c: USB new device connect on bus1/1, assigned device number 2
usb.c: kmalloc IF 8cf1f280, numif 1
usb.c: new device strings: Mfr=1, Product=2, SerialNumber=3
usb.c: USB device number 2 default language ID 0x409
Manufacturer: TEAC
Product: IBM USB CD-ROM Drive
SerialNumber: 0000000009000571
scsi0 : SCSI emulation for USB Mass Storage devices
  Vendor: IBM       Model: USB CD-ROM        Rev: 20B4
  Type:   CD-ROM                             ANSI SCSI revision: 02
Detected scsi CD-ROM sr0 at scsi0, channel 0, id 0, lun 0
sr0: scsi3-mmc drive: 10x/10x cd/rw xa/form2 cdda pop-up
WARNING: USB Mass Storage data integrity not assured
USB Mass Storage device found at 2
usb.c: usb-storage driver claimed interface 8cf1f280
VFS: Disk change detected on device sr(11,0)
ISO 9660 Extensions: RRIP_1991A
#
```

〔リスト7〕CD-ROMドライブのデバイスのようす

```
# mount /dev/scd0 /mnt
mount: block device /dev/scd0 is write-protected, mounting read-only
# df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda1              3952852     91384   3660660   2% /
/dev/scd0               380928    380928         0 100% /mnt
#
```

● SCSIデバイスの選択

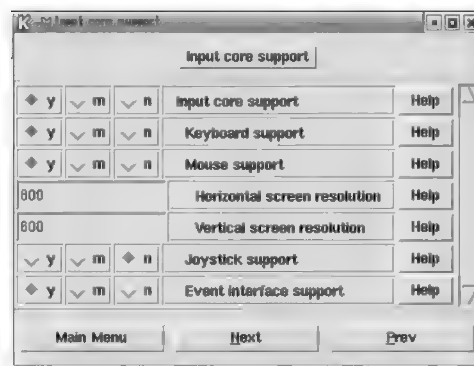
USBでCD-ROMドライブを接続するには，実はSCSIとしての設定も必要です．USBなのになぜSCSIが関係するのか？と思われるかもしれませんが，USB経由のHDDやCD-ROMドライブなどのデバイスは，SCSIのエミュレーションという形で実装されているからです．そこで，CD-ROMドライブを接続する場合はSCSIデバイスのCD-ROMドライブを接続するという設定をします．

選択後のメニュー画面を図3に示します．

● 入力デバイスの選択

キーボードやマウスを使う場合は，“Input core support”から“Keyboard”および“Mouse”を選択します．選択後のメニュー画面を図4に示します．

〔図4〕キーボードやマウスの選択




## 3 動作確認

カーネルコンフィグレーションが終了したら，カーネルの再構築をし，新しく作成したvmlinuxを実機に転送して，再構築したカーネルを立ち上げます．

● USBハブ

USBハブは，カーネルにUSBを組み込んだ時点でUSBハブ用のドライバも組み込まれるので，特別それを意識してドライバを組み込む必要はありません．とくに問題なく普通に使えます．

● CD-ROMドライブのアクセス

一般的にSCSIでの接続の場合，CD-ROMドライブはsr0，sr1…，もしくはscd0，sdc1…，で定義されます．どちらも実体は同じ場合が多いようです．今回のUSBでの接続の場合は，リスト6によるとデバイス名がsr0になります．しかし本評価キットでは，scd0…のみが定義され，sr0…は定義されていなかったので，scd0でデバイスを接続しました．コンソールの状態をリスト7に示します．

● USBフラッシュメモリスティックとHDD

接続時のメッセージをリスト8に示します．この場合，ボリューム名がsda，パーティションが一つ，sda1となります．パーティションを切り直すのであれば，

```
fdisk /dev/sda
```

フォーマットするのであれば，

〔リスト8〕
HDD接続時のようす

```
# hub.c: USB new device connect on bus1/1, assigned device number 6
Manufacturer: USB
Product: Solid state disk
SerialNumber: 230760A43EBA2BAA
scsi2 : SCSI emulation for USB Mass Storage devices
  Vendor: BUFFALO   Model: ClipDrive          Rev: 1.11
  Type:   Direct-Access                      ANSI SCSI revesion: 02
Detected scsi removable disk sda at scsi2, channel 0, id 0, lun 0
SCSI device sda: 64512 512-byte hdwr sectors (33 MB)
sda: Write Protect is off
 sda: sda1

#
```

〔リスト9〕
キーボード接続時

```
hub.c: USB new device connect on bus1/2, assigned device number 4
Manufacturer: STRONG MAN
Product: STRONG MAN USB K/B
event0: Event device for input0
keybdev.c: Adding keyboard: input0
input0: USB HID v1.00 Keyboard [STRONG MAN STRONG MAN USB K/B] on usb1:4.0
usb.c: hid driver claimed interface 8c261f40
event1: Event device for input1
keybdev.c: Adding keyboard: input1 ←── Ⓐ
input1: USB HID v1.00 Device [STRONG MAN STRONG MAN USB K/B] on usb1:4.1
usb.c: hid driver claimed interface 8c261f58
event2: Event device for input2
mouse0: PS/2 mouse device for input2
input2: USB HID v1.00 Mouse [STRONG MAN STRONG MAN USB K/B] on usb1:4.2



hub.c: USB new device connect on bus1/1, assigned device number 5
Manufacturer: NOVATEK
Product: USB Mouse STD.
event3: Event device for input3
mouse1: PS/2 mouse device for input3
input3: USB HID v1.00 Mouse [NOVATEK        USB Mouse STD.  ] on usb1:5.0
#
```

```
mkfs /dev/sda1
```

とします．通常のディスクと扱いは同じです．

USBフラッシュメモリスティックも，ソフトウェア的に見るとHDDとまったく同じ扱いです．なお，一つのUSB機器で複数のメモリカードスロットをもつメモリカードリーダ/ライタの場合，うまく認識しないものがありました．

● キーボードとマウス

USBキーボード接続時のメッセージをリスト9に示します．リスト中のⒶのメッセージからわかるように，USBケーブルをつなぐだけで使用可能となります．その結果，この評価キットの場合は，二つのキーボードが同時に使える状態になります．

本評価キットには，Microwindowsによる GUIのサンプルも添付されており，PS/2マウスを使えばマウスカーソルも動きます．PS/2ポートに接続したマウスのデバイス名は，本評価キットではpsauxとなっていて，それをmouseというデバイス名にしてアプリケーション上から制御しています．USBマウスを接続すると，デバイス名はusbmouseとして認識されるので，たとえば，

```
rm mouse
ln -s usbmouse mouse
```

などと操作して，usbmouseをデバイス名mouseとして定義すれば，USBマウスでGUIを操作することができます．

## まとめ

PCIバスとキャッシュがらみの不具合を修正し，USBを動かしてみました．USB関連のドライバは修正なしで動作しました．使用しているカーネルは2.4.5で，サポートされているデバイスはあまり多くありませんが，最新バージョンになると，かなり追加されています．

組み込み機器としてUSBを実装しておくことは，保守や拡張性に関しても，かなり有効ではないかと思われます．

さかわ・のぶひろ




第25回(最終回)

# 総集編――いままで解説してきた開発環境/ツールの最新情報

水野 貴明

これまで2年以上にわたって続けてきたこの開発環境探訪も，いよいよ今回で最終回となった．そこで今回は，これまで解説してきたさまざまな開発環境やツールを順に振り返り，紹介したときから現在までのそれぞれの開発環境の状況の変化などを追ってみる．解説してきたものを表1に，一覧形式でまとめた．



## ActiveBasic

ActiveBasicは，N88-BASICという，かつての国民機であったPC-9801シリーズに搭載されていた言語との互換性を重視しながらも，GUIへの対応なども行っているBASIC言語である．N88-BASICのプログラムすべてがそのまま動くというわけではないものの，N88-BASICにかなりよく似た書式でのプログラミングが可能となっている．

さて，ActiveBasicは，本連載で紹介した時点から現在までの間に，非常に大きな変化を遂げている．もっとも大きな変化は，インタプリタからネイティブコンパイラへと実行形態が変わったことである．そのおかげで処理速度も向上し，DLLの作成も可能になった．また，Win32APIの呼び出しなども可能になり，さまざまなWindowsアプリケーションの作成を行える．

〔表1〕紹介した開発ツールの一覧

| | 名　前 | URL | 紹介号 | 紹介時のバージョン | 現在のバージョン |
|---|---|---|---|---|---|
| 1 | ActiveBasic | http://www.discover-soft.com/ | 2001/07 | 2.2 | 3.04 |
| 2 | Aqua | 公開停止 | 2001/08 | 0.74b | ― |
| 3 | GlueX | http://www.vsa.co.jp/ | 2001/10 | 0.92 | 0.92 |
| 4 | Yabasic | http://www.yabasic.de/ | 2001/11 | 2.701 | 2.730 |
| 5 | Perl/Tk | http://www.personal.u-net.com/~ni-s/ | 2002/01 | 800.023 | 804.025（beta） |
| 6 | Plua | http://netpage.em.com.br/mmand/plua.htm | 2002/02 | 1.0b9 | 1.0 |
| 7 | REBOL/View | http://www.rebol.com/ | 2002/03 | 1.2.1 | 1.2.1 |
| 8 | ひまわり | http://hima.chu.jp/ | 2002/04 | 1.20 | 1.82d |
| 9 | UWSC | http://www002.upp.so-net.ne.jp/umiumi/ | 2002/05 | 2.4a | 2.8b |
| 10 | ScriptBasic | http://www.scriptbasic.com/ | 2002/06 | 1.0build28 | 1.0build30 |
| 11 | My Inno Setup Extensions | http://isx.wintax.nl/ | 2002/08 | 2.0.18 | 4.0.9（Inno Setup） |
| 12 | BrainF＊ck | http://www.catseye.mb.ca/esoteric/bf/ | 2002/09 | ― | ― |
| 13 | CamelBones | http://www.dot-app.org/ | 2002/11 | 0.2 | 0.3pre3 |
| 14 | SWIG | http://www.swig.org/ | 2002/12 | 1.3.14 | 1.3.19 |
| 15 | GNU indent | http://home.hccnet.nl/d.ingamells/beautify.html | 2003/02 | 2.2.8a | 2.2.9 |
| 16 | BCX | http://bcx.basicguru.com/ | 2003/03 | 3.00 | 4.25 |
| 17 | Open Perl IDE / Perlを始めよう！ | http://hp.vector.co.jp/authors/VA010286/ | 2003/04 | 1.0 / 2.0.3.6 | 1.0 / 2.0.5.0 |
| 18 | GNU GLOBAL | http://www.gnu.org/software/global/global.html | 2003/05 | 4.5.1 | 4.6.1 |
| 19 | Konfabulator | http://www.konfabulator.com/ | 2003/06 | 1.0.2 | 1.5.2 |
| 20 | Scriptol | http://www.scriptol.com/ | 2003/07 | 3.4 | 3.9 |
| 21 | Icon | http://www.cs.arizona.edu/icon/ | 2003/09 | 9.4.2 | 9.4.2 |
| 22 | TTSneo | http://hp.vector.co.jp/authors/VA021321/ | 2003/10 | 0.74 | 0.936.902 |
| 23 | D言語 | http://www.digitalmars.com/d/index.html | 2003/11 | 0.69 | 0.75 |
| 24 | Pike | http://pike.ida.liu.se/ | 2004/01 | 7.4.27 | 7.4.28 |

紹介当時は開発環境である ProjectEditor もテキストエディタとしての機能しかもっていなかったが，現在ではウィンドウのデザインからデバッグまで行う IDE へと進化を遂げた．現在でも N88-BASIC に近いスタイルでのプログラミングも可能だが，それはもはや，あくまで ActiveBasic に用意されているいくつかのスタイルの一つにすぎず，主たるプログラミングスタイルは，ウィンドウをデザインしてイベントごとにコードを記述するイベント駆動型のスタイルになり，N88-BASIC よりも Visual Basic 6 などに非常に近い印象を受ける．

紹介した当時，ActiveBasic の欠点として，実行速度の遅さ，DLL などの呼び出しなどの拡張性がないこと，ネットワーク機能がないことの3点を欠点としてあげていたが，これらはすべて改善されたといっていいだろう．今後のさらなる進歩も楽しみな言語である．



## Aqua

Aqua は，PHP4で利用されているスクリプトエンジンである Zend を用いた Windows 用のスクリプト言語で，キーボードやマウスの自動化や COM のサポートなど，Windows ならではの機能も兼ね備えていたものだった．

残念ながら Aqua はすでに公開されていない．しかし，作者の藤本氏は PHP の開発などにも参加されているようで，藤本氏のサイト( http://nx.eth.jp/)には，PHP や Zend エンジンのアップデート情報などが掲載されている．それによれば，現在ベータ版が公開中の PHP 5.0に搭載されている Zend 2.0では，private/protect/final といった属性がメンバ変数・関数に設定できるようになったり，イテレータが用意されたりと，オブジェクト指向言語としての機能に強化が図られているようだ．



## GlueX

GlueX は，ActiveX や HTML で作成した部品を，JScript( JavaScript の Microsoft 社の実装)で連携することで，Web サイトで用いられているしくみをそのまま使ったアプリケーション開発ができるシステムである．HTML で作成された部品は Scriptlet と呼ばれ，さまざまな Scriptlet が標準で用意されているほか，自分で作った Scriptlet を利用することも可能になっている．

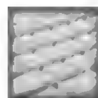
本連載で紹介してから約2年の間，GlueX は残念ながらバージョンアップされていないが，オリジナルのブラウザを作りたい，ちょっとしたツールを JavaScript で書きたいといった場合には，十分に役立ってくれる開発環境である．



## Yabasic

Yabasic は，ドイツのプログラマ Marc-Oliver Ihm 氏によって開発が行われているシンプルな BASIC 言語である．コンソールベースの実行環境だが，GUI を利用したプログラムにも対応している．同じプログラムを Windows と Linux のどちらでも動作させられる．また，ヨーロッパとオーストラリアで販売されている PlayStation 2には，この Yabasic の PS2版がついてくる．テレビ映像方式の違いから，日本で販売されている PS2では残念ながら動作しないようだが，ゲーム機に移植された BASIC 言語というのはなかなかおもしろい．

Yabasic はその後ゆるやかなバージョンアップを続けているが，今のところバグフィックスが中心で，紹介以後それほど大きな機能追加などは行われていない．しかし，先日 Web サイトに公開されたニュースによると，今後インタプリタとプログラムをまとめて，スタンドアロンの実行ファイルを作製できる機能などが追加される予定だそうだ．



## Perl/Tk

Perl/Tk は，Perl に GUI を実装するモジュールである．これはもともと Tcl という別のスクリプト言語のために作られた，Tk という GUI モジュールを Perl に移植したものだ．Perl/Tk では，ウィンドウ，ボタン，メニューといった各要素はすべてウィジェットというオブジェクトとして管理される．また，GUI 部品の画面や位置はジオメトリマネージャという，Java におけるレイアウトマネージャと同様の機能をもつシステムによって管理され，イベント駆動型のプログラミングを行うことになる．強力だが GUI の機能をもたない Perl に GUI を追加する Perl/Tk は，専門の書籍が何冊も発行されているほど有名なモジュールになっている．

Perl/Tk は，Perl 5.8の登場にあわせて，文字コードとして UTF-8を利用するように修正が行われ，現在に至っている．



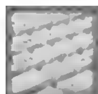
## Plua

Plua は，Lua というスクリプト言語を PalmOS に移植したものである．Lua は，PASCAL や BASIC と似た言語仕様で，関数が複数の戻り値を取れるなどの特徴をもった言語である．ブラジルのリオデジャネイロ・カトリカ大学で開発されている．Plua は，Lua をベースに GUI を構築する機能などを追加し，メモ帳や DOC 形式など，Palm 上でスクリプトをぱっと書き，そのまま実行できる点が特徴的だった．プログラムはコンパイルされ，Palm の実行ファイル形式である PRC ファイルが生成されるが，実行の際にはランタイムライブラリが必要となる．

紹介時にはベータ版だった Plua も，すでに完成版のバージョン 1.0が公開されている．当時作者の Marcio Migueletto de Andrade 氏にインタビューしたところ，シェアウェアとしてリリースされる予定だという話だったが，最終的にはフリーソフトウェアとしての公開となった．また，当時はまだ公開されるという予告がされているだけだった Plua のランタイムライブラリ「PluaRT」も，いっしょに配布されるようになってい

る．言語仕様としては，基本的にLua 4.0ベースであることに変化はないが，スライダやポップアップメニューなど，当時は使えなかったGUI部品も使えるようになったり，クリップボードへのアクセスが可能になるなど，さまざまな改良が施されている．

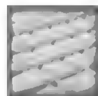
さらに，Windows用とLinux用のコンパイラが配布されるようになった点も注目される．これらのOS上でPluaのプログラムをコンパイルして，PRCファイルを作成できるのである．しかし実行(とデバッグ)には，PalmOSを実行できる環境(Palm実機やエミュレータ)が必要となる．



## REBOL/VIEW

REBOLは米REBOL Technologies社によって開発されており，WindowsやLinux，MacOSをはじめ，BeOSなども含め44もの幅広いプラットホームに対応しているスクリプト言語だ．REBOLは，コンソールベースのREBOL/Coreを中心に，GUIを追加したREBOL/VIEWやREBOL/VIEW/PRO，サーバサイドアプリケーション向けの機能を強化したREBOL/Commandなど，いくつかのパッケージが存在している．REBOLは，プログラムを大かっこ[ ]で囲んだ「ブロック」の集合体として記述するという言語仕様をもつ．また，HTTPやFTPなど14のインターネットプロトコルに対応し，インターネット上に置いたプログラムやデータファイルをローカルに置いたデータと同じように操作できたり，ネットワーク上のデータを特別に意識しなくても利用できる機能(REBOLではX-Internetと呼ぶ)が大きな特徴となっている．たとえばREBOL/VIEWでは，REBOL Desktopという実行環境が付属し，ここではインターネット上に置いたREBOLのプログラムファイルを，まるでローカルに存在しているかのようにそのまま実行することができる．

さて，紹介以後，スクリプトエンジンであるREBOL/COREのバージョンアップが行われ，2003年8月には最新版の2.5.6が公開されているが，REBOL/VIEW自体はバージョンアップがされていない．これは，REBOL Technologies社が現在，REBOL IOS(Internet Operating System)の開発に力を注いでいるからのようだ．REBOL IOSは，REBOL/VIEWやREBOL/Commandの機能をベースに，ユーザー管理やログの記録機能を追加したもので，予定表やインスタントメッセージ，カレンダ機能などがあらかじめ用意されていて，グループウェアとして利用できるものだ．もちろん，REBOLスクリプトを利用して，拡張することも可能になっている．REBOLのGUIは日本語表示ができないため，日本ではほとんどREBOLの話題が聞かれないのが残念である．ちなみにREBOL/VIEWに付属するREBOL Desktopは，2003年6月にオープンソースとなり，ソースが公開された(もちろんこれもREBOLスクリプトで記述されている)．



## ひまわり

ひまわりは，「(1+1)と、表示。」といったような，日本語でプログラミングを行うことができる言語である．HTTPやFTP，メールなどのネットワーク関係の命令や，ほかのアプリケーションとの連携機能などが強化され，単純作業の自動化や効率化に役立つよう設計されているのが特徴だ．

統合開発環境として「ひまわりエディタ」がついてきており，ステップ実行などのデバッグ機能も用意されている．テンプレートなども用意されており，開発のサポートはかなり充実している．

ひまわりではGUIを構築することもできる．GUI部品の配置はスクリプト内で行うことになるが，ひまわり自身で記述されたフォームデザイナを使って，GUIを構築するスクリプトを自動生成することができる．また，配列変数が内部的にはCSVデータとして格納されており，CSVデータを非常に簡単に扱うことができるようになっている点も興味深い．

さて，ひまわりはバージョンアップが頻繁に行われている．紹介時にも，たいへんな勢いでバージョンアップされていたが，現在でも数日おきにバージョンアップが行われることもしばしばで，開発者であるクジラ飛行机氏のひまわりに対する情熱を感じることができる．紹介後，ひまわりはXMLやデータベース接続など，さまざまな機能に対応しており，さまざまな分野に対応できるべく成長を続けている．

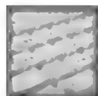


## UWSC

UWSCは，Windowsのキーボードやマウスの操作を自動化するツールである．実際にマウスやキーボードを動かしてそれを記録できるほか，BASICに似たスクリプト言語を利用することで，制御構造などを利用し，より柔軟な設定を行える．たとえば，特定のアプリケーションのウィンドウをすべて同じサイズにそろえたり，IMEの動作を監視して，常に日本語入力がONの状態に固定するといった動作を行える．

また，`FUKIDASI`という命令を使ってユーザーにメッセージを表示したり，`PRINT`文でログを出力するといったことも簡単にできる．スケジュール機能で一定の間隔でスクリプトを走らせることも可能だ．

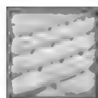
現在は，紹介時の機能に加えて，CPU使用率の計測などの組み込み関数の追加をはじめ，DLLを呼び出す機能や，マルチディスプレイへの対応など，さらに機能が強化され，使いやすくなっている．紹介時にサンプルスクリプトとして現在開いているフォルダをすべて閉じるスクリプトを作成したが，ついついフォルダをたくさん開いてしまう筆者にとってはかなり便利で，今でもよく利用している．



## ScriptBasic

ScriptBasicは，ハンガリーのプログラマPeter Verhas氏が開発しているBASIC言語だ．「Script」という名前が示すとおり，ハッシュ機能や動的配列，変数の型が自動変換されるなど，スクリプト言語としての特徴を兼ね備えている．とはいえ，言語仕様としては，非常に伝統的な雰囲気をもっており，習得するのも比較的簡単である．またScriptBasicは，外部モジュールを使って拡張が行えるようになっており，CGIやデータベース接続やzlib圧縮，グラフィックス関係などさまざまなモジュールが利用できる．Visioという GTK+を利用したGUIモジュールを使って，GUIアプリケーションを作成することも可能である．

紹介時，すでにScriptBasicには，Cへの変換機能が実装されており，変換したプログラムを，ランタイムライブラリと一緒にビルドすることで，スタンドアロンアプリケーションを作ることができた．さらに最新版のBuild30では，Cに変換することなく（つまりCのコンパイラを用意しなくても），スタンドアロンの実行ファイルを作成できるようになり，よりプログラムの配布が簡単になっている．

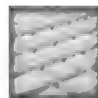


## My Inno Setup Extensions

My Inno Setup Extensionsは，Windows用のインストーラ作成ツールだ．正確にいうと，Inno Setup（http://www.jrsoftware.org/isdl.php）というオープンソースのインストーラ作成ツールがあり，それにPascalスクリプトと呼ばれる，Pascalライクなスクリプト言語の実行環境をつけたものがMy Inno Setup Extensionsなのである．Inno Setupでは，インストーラの定義にはテキストファイルが用いられる．テキストファイルは[Setup]，[Messages]といったセクションに分けられており，そこにインストール先や，インストールすべきファイルなどを指定し，最後にコンパイルを行うことで，単体のインストーラを作成できるしくみだ．My Inno Setup Extensionsでは，[Code]というセクションを追加でき，そこにスクリプトを埋め込める．スクリプトは，インストーラの初期化の際や，ページが切り替わったときなど，決められたタイミングで呼び出すことができる．また，他のセクションでのパラメータ設定の際に関数を呼び出して，パラメータを動的に設定することなども行える．

スクリプトは「Innerfuse Pascal Script（IFPS）」というDelphiの言語仕様に基づいたスクリプトエンジンが用いられており，レジストリや環境変数などのシステムの情報を調べたり，カスタムページを作成するなど，組み込み関数も充実している．

My Inno Setup Extensionsは，バージョン4より，その全機能が本家Inno Setupに取り込まれた．Inno Setup自体も，紹介時から2回のメジャーバージョンアップを経ており（My Inno Setup Extensionsのバージョンは Inno Setupに準じていた），マルチリンガル対応をはじめとして，さまざまな強化が図られている．



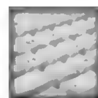
## BrainF*ck

BrainF*ckは，1993年にスイス人プログラマのUrban Mueller氏によって考え出されたプログラミング言語だ．その見た目は，たとえば次のようになる．

```
>+++++++++[<++++++++>-]<.
```

これは画面上に「H」と出力するだけのプログラムだが，ぱっと見ても，なにがなんだかまったくわからない．それもそのはず，BrainF*ckはEsoteric Programming Languages（難解プログラミング言語）と呼ばれるジャンルに属するプログラミング言語で，わかりにくいことに意義を見出している言語だからだ．したがって実用性は低いが，わかりにくい言語を使ってパズルのように組み合わせてプログラムを書いていくことは，プログラミングの楽しさを追及した究極の形の一つといえるのかもしれない．BrainF*ckは1文字からなる八つの命令セットで構成されており，チューリングマシンをプログラミング言語にしたような形をしている．したがって，どんなプログラムであっても，理論上はBrainF*ckで記述できる．

本連載では，難解プログラミング言語として同時に，オラウータン向けのプログラミング言語である「Ook（http://www.dangermouse.net/esoteric/ook.html）」や，プログラムがまるで料理のレシピのように見える「Chef（http://www.dangermouse.net/esoteric/chef.html）」も紹介した．これらも非常に興味深いが，そのほかにもWebを探せば，プログラムが抽象画のように見える「Piet（http://www.dangermouse.net/esoteric/piet.html）」や，ホワイトスペース（Tab，改行，空白）だけでプログラミングを行う「Whitespace（http://compsoc.dur.ac.uk/whitespace/）」など，さまざまなものが発見でき，難解プログラミング言語の愛好家が世界にたくさんいることがよくわかる．

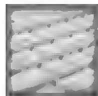


## CamelBones

CamelBonesは，Mac OS XのもつAPIセットの一つであるCocoaをPerlから使えるようにするフレームワークだ．これを使うことで，PerlでMac OS XのGUIアプリケーションが作成できるようなる．開発は，Appleが無償公開している公式開発環境であるProject Builder（Mac OS X 10.3からはXCode）およびInterface Builderを利用して行うことができる．Objective-CやJavaを利用した場合と同様に，Interface BuilderでGUIを構築し，それをPerlから操作できるようになるのだ．PerlでMac OS XのGUIアプリケーションを作成できるということは，筆者のようなPerlをよく利用しているMac OS Xユーザーには非常にありがたい．

ちなみに，スクリプト言語を利用してCocoaアプリケーション

を作成するという試みは，Perl以外でも，RubyのRubyCocoa（http://www.imasy.or.jp/~hisa/mac/rubycocoa/index.ja.html），PythonのPyObjC（http://pyobjc.sourceforge.net/）なども登場しており，自分の好みのスクリプト言語でCocoaアプリケーションが書ける環境が整いつつある．また，Perlの場合は，Mac OS Xの10.2以降では標準で「PerlObjCBridge.pm」というPerlからCocoa APIにアクセスするためのモジュールがつくようになっており，こちらを利用しても，PerlでCocoaアプリケーションを記述できる．



## SWIG

SWIGはSimplified Wrapper and Interface Generatorの略で，C/C++のコードを，PerlやPHPなどの各種言語から呼び出せるように，間をつなぐラッパコードを記述してくれるツールである．昔作ったCの関数をPerlから利用できるようにしたいとか，この処理はC/C++で書いたほうが効率的なのに…といった場合に威力を発揮してくれる．

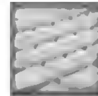
利用方法は非常に簡単で，C/C++のプログラムと，インターフェースファイル（C/C++のプログラムで定義されている関数の中で，どれをスクリプト言語から利用可能にするかを指定するファイル）を用意し，SWIGに渡すだけだ．これで自動的にラッパプログラムが出力されてしまう．出力されるのは，C/C++のプログラムといっしょにコンパイルするラッパプログラム，そしてコンパイルされたプログラムに呼び出し側の言語からアクセスするためのプログラムファイルの二つである．その後，gccやVisual C++などのコンパイラでC/C++のプログラムを出力されたラッパプログラムといっしょにコンパイルすれば，呼び出し側の言語からのアクセスが可能になる．

紹介時から現在までのバージョンアップでは，1.3.14→1.3.19と数値としては小さなものだが，その間にそれまで対応していたGuile，Java，Mzscheme，OCAML，Perl，PHP，Python，Ruby，Tclのほかに，ChickenとC#への対応がなされている．また，本連載でも第24回で取り上げたPikeへの対応も進められていると，ドキュメントに記されている．



## GNU indent

GNU indentは，GNUプロジェクトが管理/開発を行っているC言語用（C++には対応していない）のソースコードフォーマッタで，1976年にBSD UNIXの一部として開発されたことに端を発する，30年近い歴史をもつソフトウェアである．このソフトウェアは，if文の条件文の後，括弧の前で改行するか否かとか，インデントにはタブを使うのか，スペースを使うのか，といった細かい指定をすることで，その指示にしたがって，ソースコードをフォーマットしなおしてくれるツールだ．コード中の改行やインデントのルールは開発者によって異なるため，自分以外の開発者が書いたソースコードは非常に見にくい場合がある．そのような場合，GNU indentを使えば，自分の見やすい形式に整形することが可能なのだ．また，GNU indentを紹介した際に，逆に「ソースコードを読みにくくする」ツールであるCOBF（http://home.arcor.de/bernhard.baier/cobf/）も紹介した．こちらも，メンテナンス程度ではあるがバージョンアップも行われており，開発は続けられているようだ．

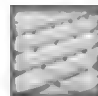


## BCX

BCXは，BASICのプログラムをCのプログラムに変換するコンバータである．変換したCのプログラムをCのコンパイラでコンパイルすることで，スタンドアロンのアプリケーションが作成できるというものだ．中間形式に変換してランタイムライブラリをリンクする形ではなく，完全なBASICからCのコードへの変換を提供する．BCXの文法は，QuickBasic，Visual Basic，PowerBasicを混ぜたものになっている．

BCXは，LCCというフリーのコンパイラを有効に活用するために開発されたものとのことで，LCCでコンパイルすることを前提としたコードを出力する．Cのコンパイラを活用するために，そのコンパイラ向けのCのコードを出力するBASICコンバータを作るというアプローチはなかなかユニークで興味深い．

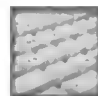
BCXは現在もかなりの速度でバージョンアップが行われており，COMのサポートや印刷機能，そのほか数多くの組み込み関数の追加などが行われている．



## Open Perl IDE/Perlを始めよう！

この回は，「Open Perl IDE」と「Perlを始めよう！」という二つのPerlの開発環境を紹介した．これらはどちらも，Perlをターゲットとし，プログラム開発から，実行，デバッグまでめんどうをみてくれる統合開発環境である．しかし，その得意分野は両者で異なり，Open Perl IDEがデバッグの簡便性を高めていることが特徴的なIDEであったのに対し，「Perlを始めよう！」は「マクロ」や「ひな形」などの機能により，プログラミング中の簡便性を高めることを重視している開発環境といえる．

Perlは汎用的で非常に優れたスクリプト言語であり，ちょっとしたテキスト処理などに活用すると，大幅に手間を軽減できる．しかし，多くのWindowsユーザーの場合，DOSプロンプトを起動してPerlのスクリプトを実行するという作業はなかなか敷居が高く，Perlの利用も難しくなっている．しかし，こうしたIDEが登場することによって，より多くの人がPerlに触れられる環境が整ってくるのではないだろうか．



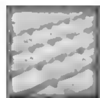
## GNU GLOBAL

GNU GLOBALは，多摩通信社（http://tamacom.com/）

が作成しているソースコードタグシステムだ．これはソースコード中のシンボル，つまり関数や変数の定義場所や実際に呼び出している場所など，さまざまな位置の情報を解析して記録しておき，後からその位置情報を利用することができるシステムのことを指す．

GNU GLOBALでは，作成したタグ情報を用いて，特定のシンボル名で検索を行ったり，シンボルの仕様場所から定義位置へとハイパーリンクをはったHTMLファイルを出力するといったことが可能になる．

他人の書いたソースコードや，自分の書いたソースコードであっても，書いてから時間が経ってしまったものなどを読まなければならない場合に，効率を向上してくれる，非常に重宝するツールなのだ．本連載で紹介した後，GNU GLOBALは，これまでのC，C++，yacc，Java，アセンブラに加えて，PHPスクリプトにも対応した．また，ソースコードをHTML化する際に，CVSリポジトリにリンクする機能も追加されている．

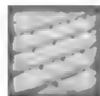


## Konfabulator

Konfabulatorは「Kaleidoscope」というMacOS 9用のデスクトップカスタマイズツールを作成したArlo Rose氏が開発しているツールで，Widgetと呼ばれるデスクトップアクセサリのMac OS X用実行環境だ．WidgetはXMLとJavaScriptで記述されており，標準でも時計やバッテリメータなどがついてくるほか，新しいWidgetを自作することもできる．また，配布元サイトにはGalleryと呼ぶ，第三者が作成したWidgetを登録できるデータベースが設置され，ゲームやRSSアグリゲータなど，世界中のユーザーが作成したさまざまなWidgetが公開されている．

紹介当時のバージョンは1.02だったが，現在は1.52にバージョンアップされた．キー入力やドラッグ&ドロップに対応し，よりバラエティに富むWidgetの作成が可能になった．また，画像をスライド表示させてWidgetにアニメーション効果をもたせるといった，演出力をアップするための機能も追加されている．

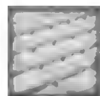
また，KonfabulatorはWindows版を公開する予定があるようで，すでにベータ版の公開用ページが用意されている（`http://www2.konfabulator.com/beta/`）．しかし，ベータ版の配布はまだ始まっていない．Windows版とMac版で，どこまでWidgetファイルに互換性が保たれるのかは不明だが，Macよりずっと数の多いWindowsユーザーを取り込むことで，よりバラエティに富むWidgetが公開されるようになることが期待できる．ただし，Windowsにはsamurize（`http://www.samurize.com/modules/news/`）といった同様の機能をもつソフトウェアがすでに存在するため，どのような形で共存していくのかが注目される．



## Scriptol

Scriptolは，PerlやPHP，LuaやBASICなどの多くの言語の特徴を取り入れて作られたスクリプト言語で，「これまでできなかったことを実現する」ということよりも「なるべくわかりやすく，簡単に物事を行う」という思想のもとに作られている言語である．Scriptolで書かれたプログラムはそのまま実行されるのではなく，C++，もしくはPHPのスクリプトとして変換されることで，実行が可能になる．C++に変換したコードはLinuxならgcc，WindowsならBorland C++か，MinGWでコンパイルできる．Windows版には簡単なIDEも付属する．

なおScriptolは，本連載での紹介後，Javaのクラスのインポートが可能になるなど，着実に進歩を遂げている．



## Icon

Iconは，1960年代にAT&T（当時）のベル研究所で開発されていたSNOBOLという言語の流れを汲み，現在はアリゾナ大学で開発が行われている言語である．Iconの言語仕様はC言語やPASCALなどと似ている部分も多いが，いくつかのユニークな特徴がある．その一つが，すべての式は「成功（Succeed）」と「失敗（Fail）」のどちらかの状態を取るという点だ．多くのプログラミング言語では，真（True）と偽（False）という論理値を使って式が正しいのかを表現するが，Iconでは，式の値とは関係なく，成功と失敗のどちらかの状態を取ることになる．そして，式の一部が失敗の状態になると，その式全体が失敗とみなされ，その実行がキャンセルされる．

Iconの持つもう一つのユニークな特徴は，ジェネレータの存在である．これは「複数の値を順番に結果としてもつことができる式」を意味する．たとえば，文字列の中から特定の文字列を探す`find`という関数は，特定の文字列が複数回発見できる場合には，発見できる回数だけ値を生成するジェネレータとして処理される．ジェネレータは，`every`という演算子とともに使うことで，すべての値が生成されるまで処理を繰り返すことができる．さらに，複数のジェネレータを組み合わせて利用すると，それらのジェネレータの出力する値のすべての組み合わせを評価できるようになる．また，ジェネレータを条件式とともに利用することで，条件式が「失敗」と判断された組み合わせが自動的にキャンセルされるので，条件分岐を書かずに，目的の組み合わせだけを取り出せるようになっている．このように，目的のデータだけに簡単にアクセスできる機能は「goal directed evaluation」と呼ばれ，Iconのもっとも大きな売りとなっている．



## TTSneo

TTSneoは，「ひまわり」と同様，日本語でのプログラミン

グが可能な開発環境である．以前は，「テクノロジーターミナルスクリプト」という名前で公開されており，ひまわりを紹介した際にもコラムでふれているが，TTSneoという新しい形に生まれ変わったということで，あらためて取り上げたものだ．TTSneoには一般的な計算処理やファイル操作のほか，GUIの利用，動画や音楽の再生，タートルグラフィック，Direct3D，FTP/HTTPによるインターネットアクセス，正規表現など，幅広い機能が用意されており，ホビーから実用的なものまで，さまざまなアプリケーション作成に対応している．また，TTSneoはLOGOの流れを取り入れており，タートルグラフィックの機能も用意されているのが大きな特徴となっている．

また，TTSneoの注目すべき特徴としては，TTSneoは若い世代，もっといえば小中学生にでも比較的容易にプログラミングが体験できるような配慮がたくさんされている点だ．マニュアルは漢字に振り仮名が振られたものも公開されていたり，非常にやさしいチュートリアルが公開されていたりと，プログラミングを学ぶうえでの，プログラミング以外の障壁がなるべく低くなるように工夫されているのだ．こういった敷居の低い言語でプログラミングの楽しさを学び，やがて他の言語も使うようになっていくというステップは，プログラミングを学ぶうえで，なかなか有効な手段の一つではないだろうか．



## D言語

D言語は，Digital Mars C++（かつてのSymantec C++）の開発者であるWalter Bright氏が開発しているオブジェクト指向言語で，C/C++，Java，C#といった既存の言語を研究し，その良いところをどんどん取り込んだ言語となっている．その特徴はD vs Other Languages（`http://www.digitalmars.com/d/comparison.html`）というページにまとめられているが，ガベージコレクションや配列の境界チェック，オペレータのオーバロードやインラインアセンブラなど，それぞれの言語の多くの特徴を併せもっていることがわかる．また，関数の中に関数をネストして記述できたり，クラスの単体テストを行う機能が用意されていたりと，ユニークな機能も数多く実装されている．

現在D言語はまだ開発中で，仕様と実装で挙動が異なったりするなどの問題もあるが，非常に速い速度でバージョンアップが行われており，それらの問題が解決するのもそれほど遠くないことだろうと予想できる．もしかすると，いずれC#やJavaと並ぶ開発環境の選択肢として，あたりまえのようにD言語が並べられる日がくるかもしれない．



## Pike

Pikeは，スウェーデンのリンシェーピン大学の情報工学科でメンテナンスされているオブジェクト指向のスクリプト言語で，もともとロールプレイングゲーム開発用の言語だったLPCという言語を基にして作られたという，ユニークな経歴をもつ．その言語仕様はC/C++によく似ている．Pikeでは，プログラムはコンパイラによって一度バイトコードに変換されてから実行されるが，その前にプリプロセッサによって処理が行われる．プリプロセッサでは，条件付きコンパイルを行えるが，「`#pike`」ディレクティブでバージョンを指定すると，古いバージョンをエミュレートできるという機能がおもしろい．

モジュールを使って機能を拡張する機能もある．GUIやネットワークアクセスを行う機能がモジュールとして用意されているほか，PikeやC言語を使って新しいモジュールを作ることも可能だ．連載第14回で紹介したSWIGも，Pikeのモジュール作成に対応する予定になっており，今後，よりモジュールが作成しやすい環境が整うことだろう．

まだまだドキュメントの不備が目立つなど，開発途上の言語だが，もともとPikeはRoxen Web ServerというWebサーバを書くために開発が続けられており，そのWebサーバもきちんと公開されている．このことからも，実用に耐え得る言語であることは実証されているといってよいだろう．

## おわりに

これまで24回にわたり，さまざまなプログラミング言語やその周辺ツールについてみてきた．まとめて振り返ってみると，いろいろなツールがあるものだと，あらためて思わされる．もちろん，現在公開されている開発ツールの数は，この程度であるはずもなく，本連載は無数にあるツールのうちの，ほんの一角を切り取ったにすぎないが，それでもさまざまなタイプのツールを紹介できたのではないかと思っている．

この連載を通して筆者があらたに思ったのは，紹介した開発ツールにはそれぞれすべて，長所と短所，向き不向きがあり，時と場合によって使い分けることで，より効果的に利用することができるということである．一つの開発ツールに固執せず，臨機応変に利用するツールを変えていくことで，より効率的な開発が可能になるはずだ．

もし，ここで紹介したツールに少しでも興味をもったのであれば，ぜひ実際に自分で使ってみることをおすすめする．新しい開発ツールを使うことは，非常におもしろい体験である．そして，おもしろいだけでなく，開発の新しい可能性にも気づかせてくれる．本連載を通じて，筆者はそのことをつくづく感じさせられた．本稿が，そのことを読者の皆さんに少しでも伝えられていれば，と願っている．

みずの・たかあき

初級ドライバ開発者のための

# Windowsデバイスドライバ開発テクニック

## 第6回（最終回） DriverStudioを使ったドライバ開発事例

丸山 治雄

DDKを使い，ドライバをいきなり一から開発するのは荷が重いというときのために，ドライバの開発をサポートするソフトウェアが販売されています．代表的なものとして，ドライバのスケルトン（骨格）のソースコードを生成してくれるDriverStudio〔日本コンピュウェア（株）〕と，ソースコードは出力されないのですが，必要な機能を組み込むことができるWinDriver〔エクセルソフト（株）〕があります．

DriverStudioは，すべてのソースコードを自動的に作成してくれるので，作成されたソースコードに必要な機能を追加すれば目的のドライバが作成できます．一方，WinDriverは，ドライバの核の部分はソースコードを作成せずに，追加したいソースコードをプラグインのように追加することで，目的のドライバを作成することができます．ちなみに，ここで想定しているハードウェアであるKIT 1050 PLX Getter II〔（株）ケーアイテクノロジー〕用のドライバも，DriverStudioで作成されています．

この連載では，ドライバの構造を学習することが目的なので，ソースコードをすべて確認できるDriverStudioのバージョン2.6を使用して，KIT 1050ボード用のドライバを開発するまでを解説します．英語版はバージョン3.0が発表されています．筆者も現在評価中ですが，とくに大きな問題は見つかっていません（古いバージョンではコラムのような問題もある）．

なお，DriverStudioはドライバ開発をサポートするツールですが，これ単体でドライバが作成できるわけではありません．あくまで開発サポートツールなので，別途DDKとコンパイラは必要です．DriverStudioで生成されるソースコードはC++のみなので，C++対応のコンパイラが必要になります．今回はMicrosoft社のVisual Studio 6.0を使用しました（.NETは現在評価中）．

### 6.1 インストールとジェネレーション

DriverStudioのインストールはウイザードの指示に従い行うだけなので，特段の注意点はありません．あえて挙げるなら，DriverStudioをインストールする前に，コンパイラとDDKをインストールしておく必要があります．また，インストール後のDriverStudio用のライブラリ作成では，必要なものをビルドしてください．

インストールが完了した後，コンパイラ（Visual Studio 6.0）を起動すると，DriverStudioのアイコンが登録されているので，そのアイコンをクリックするとドライバソースコードの作成ウイザードが表示されます（図6.1）．ここでプロジェクト名を指定します．

次にドライバのタイプを指定します．Windows NT 4.0は，Windows NT専用ドライバ作成のときに指定します．プラグ＆プレイに対応したドライバを作成する場合は，WDMタイプのドライバを指定してください．今回はWDMで作成することにします（図6.2）．

次にドライバのバスを選択します．バスとしてPCIを指定するとベンダIDとデバイスIDの入力部分が表示されるので，該当PCIデバイスのベンダIDとデバイスIDを入力します（図6.3）．次にドライバクラスを問い合わせてきますが，表示されている内容のままで次に進みます．

次にドライバ内で処理する項目を選択します．図6.4のように必要な項目をチェックします．ここで，「デバイスコントロール」をチェックすると，ウイザードのステップ9/10で詳細なコードの定義が行えます．

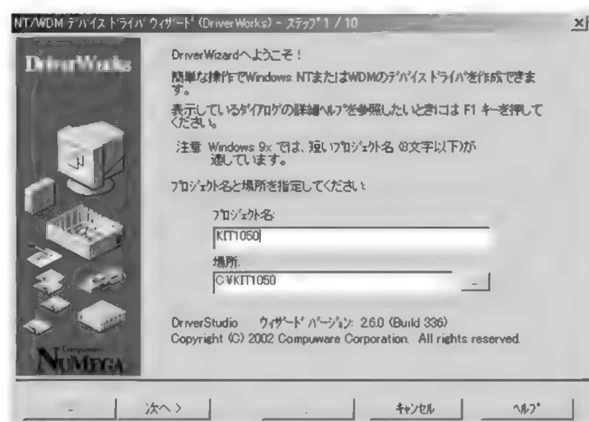
次にアプリケーションからの要求をキューイングするか否かの設定を行います．これは，たとえばアプリケーションからリード要求がきたときに，要求を一度キューイングして受け付け順に処理を行うというような場合に使用します（図6.5）



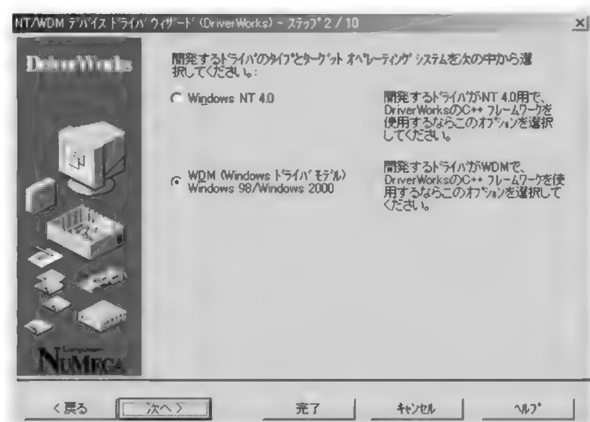

### DriverStudioのバージョン2.6.0の場合

DriverStudioのバージョン2.6.0をWindows 2000のService Pack 4の環境にインストールすると，問題が発生します．日本コンピュウェア（株）から，パッチプログラムがリリースされているので，必ずパッチを当てるようにしてください．Windows XPでは問題ありませんでした．

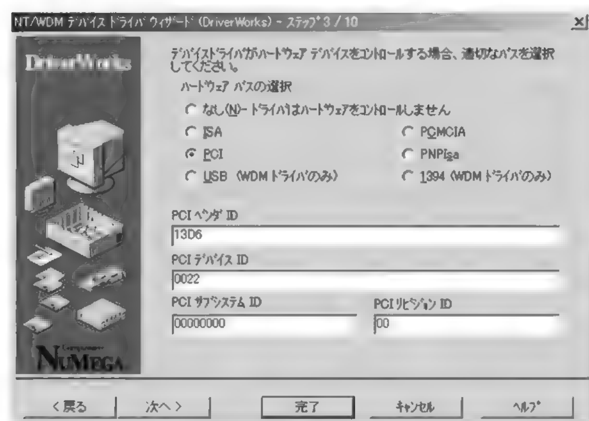
〔図6.1〕プロジェクト名の指定



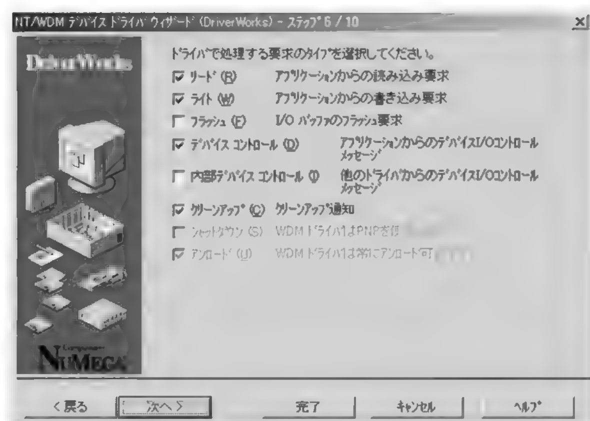

〔図6.2〕ドライバタイプの指定



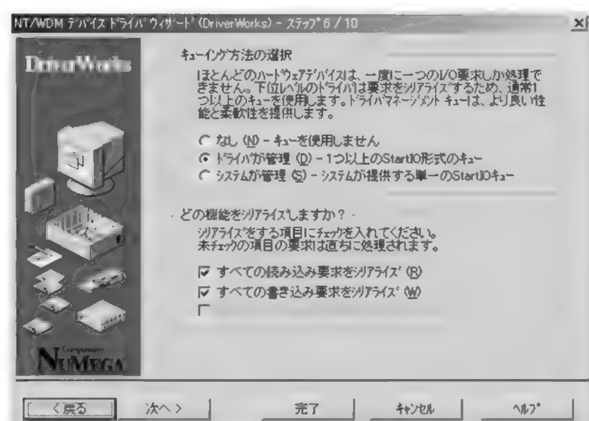

〔図6.3〕バスの指定/ベンダID＆デバイスIDの指定




〔図6.4〕ドライバ内で処理する項目の選択



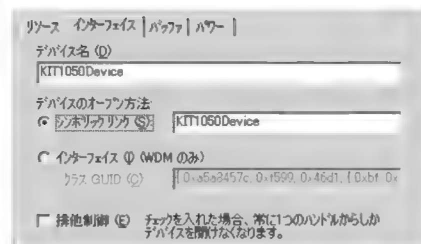

〔図6.5〕キューイングの設定




〔図6.6〕PCIボードのリソース指定




〔図6.7〕デバイスオープン方法の設定




次にドライバを起動するときにレジストリから値をロードするパラメータを設定します．ロードするパラメータがないときはスキップしてください．ここで設定した値は，ドライバ登録時に使用するINFファイルにも反映されます．

次にPCIボードのリソースを指定します．KIT1050ボードは，メモリ空間を二つ，I/Oポートを一つ，割り込み（IRQ）を一つ使うので，そのすべてを指定します（図6.6）．また同じステップのインターフェースにデバイスのオープン方法があります．標準では「クラスGUIDを使用する」になっています．クラスGUIDはわかりにくいので，シンボリックリンクを使用することをお勧めします（図6.7）．シンボリックリンクとクラスGUID

〔リスト 6.1〕シンボリックリンクとクラス GUID によるドライバオープンの違い

```
// Name used to open device
//
char *sLinkName = "¥¥¥¥.¥¥KIT1050Device0";

int __cdecl main(int argc, char *argv[])
{
    hDevice = OpenByName();
    if (hDevice == INVALID_HANDLE_VALUE)
    {
        printf("ERROR opening device: (%0x) returned from
                        CreateFile¥n", GetLastError());
        Exit(1);
    }
    else
    {
        printf("Device found, handle open.¥n");
    }
}

////////////////////////////////////////////////////////////////
// OpenByName
//
//      Open a handle to the requested device
//
HANDLE OpenByName(void)
{
    // Create a handle to the driver
    return CreateFile(sLinkName,
                        GENERIC_READ | GENERIC_WRITE,
                        FILE_SHARE_READ,
                        NULL,
                        OPEN_EXISTING,
                        0,
                        NULL);
}
```

（a）シンボリックリンクによるドライバオープン

```
#define KIT1050Device_CLASS_GUID ¥
 { 0x19b3d183, 0x52c, 0x4ed2, { 0x95, 0xd4, 0x98, 0xb5, 0xd6,
                                0x36, 0xfa, 0x70 } }

GUID ClassGuid = KIT1050Device_CLASS_GUID;
int __cdecl main(int argc, char *argv[])
{
    hDevice = OpenByInterface( &ClassGuid, 0, &Error);
    if (hDevice == INVALID_HANDLE_VALUE)
    {
        printf("ERROR opening device: (%0x) returned from
                       CreateFile¥n", GetLastError());
        Exit(1);
    }
    else
    {
        printf("Device found, handle open.¥n");
    }

}

// OpenByInterface
//
// 与えられたインターフェイス クラスで見つかった nth デバイスを開きます.

HANDLE OpenByInterface(
        // インターフェイスクラスを識別する GUID を指します.
        GUID* pClassGuid,
        // 列挙されたデバイスを開くためのインスタンスを指定します.
        DWORD instance,
        // エラー ステータスを受信する変数のアドレス
        PDWORD pError
        )
{
    HANDLE hDev;
    CDeviceInterfaceClass DevClass(pClassGuid, pError);

    if (*pError != ERROR_SUCCESS)
        return INVALID_HANDLE_VALUE;

    CDeviceInterface DevInterface(&DevClass, instance, pError);

    if (*pError != ERROR_SUCCESS)
        return INVALID_HANDLE_VALUE;

    hDev = CreateFile(
        DevInterface.DevicePath(),
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        NULL
        );

    if (hDev == INVALID_HANDLE_VALUE)
        *pError = GetLastError();

    return hDev;
}
```

（b）クラス GUID によるドライバオープン

〔リスト 6.2〕シンボリックリンクとクラス GUID によるデバイス作成の違い

```
NTSTATUS KIT1050::AddDevice(PDEVICE_OBJECT Pdo)
{
    KIT1050Device * pDevice = new (
             static_cast<PCWSTR>(KUnitizedName(L"KIT1050Device", m_Unit)),
             FILE_DEVICE_UNKNOWN,
             static_cast<PCWSTR>(KUnitizedName(L"KIT1050Device", m_Unit)),
             0,
             DO_DIRECT_IO
                 | DO_POWER_PAGABLE
             )
        KIT1050Device(Pdo, m_Unit);
}
```

（a）シンボリックリンクによるデバイス作成

```
NTSTATUS KIT1050::AddDevice(PDEVICE_OBJECT Pdo)
{
    KIT1050Device * pDevice = new (
             static_cast<PCWSTR>(KUnitizedName(L"KIT1050Device", m_Unit)),
             FILE_DEVICE_UNKNOWN,
             NULL,
             0,
             DO_DIRECT_IO
                 | DO_POWER_PAGABLE
             )
        KIT1050Device(Pdo, m_Unit);

}
```

（b）クラス GUID によるデバイス作成

〔リスト 6.3〕DriverEntry 処理ルーチン（KIT1050.CPP）

```
// KIT1050.cpp
//
// DriverWizard バージョン DriverStudio 2.6.0 (Build 336) に
// よって生成されました.
// Compuware の DriverWorks クラス
//

#define VDW_MAIN
#include <vdw.h>
#include "KIT1050.h"
#include "KIT1050Device.h"

#pragma hdrstop("KIT1050.pch")

// DriverWizard バージョン DriverStudio 2.6.0 (Build 336) に
// よって生成されました.

// デフォルト 32ビットタグ値を new によってアロケートされた
// 各ヒープ ブロックに設定します. BoundsChecker を使い,
// メモリ プールを表示します.
// この値は, グローバル関数 SetPoolTag()の使用によってオーバライドされます.
POOLTAG DefaultPoolTag('1TIK');

// グローバルドライバトレースオブジェクトを生成します.
// TODO:    トレースメッセージをデバッグ ビルドでだけ表示したい場合は,
//          KDebugOnlyTrace を使います. トレースメッセージを常に
//          表示したい場合は Ktrace を使います.
KTrace t("KIT1050");

//////////////////////////////////////////////////////////////////
// INIT セクションの開始
#pragma code_seg("INIT")

DECLARE_DRIVER_CLASS(KIT1050, NULL)

//////////////////////////////////////////////////////////////////
//  KIT1050::DriverEntry
//  ルーチン ディスクリプション:
//      ドライバがロードされると, システムによって呼び出される
//      最初のエントリ ポイントです.
//  パラメータ:
//      RegistryPath - レジストリ内のドライバパラメータを参照するために
//          使用する文字列. KIT1050 を配置して, こちらを検索してください.
//          HKEY_LOCAL_MACHINE¥SYSTEM¥CurrentControlSet¥
//                                                  Services¥KIT1050
//  戻り値:
//      NTSTATUS - エラーが起きなかった場合は, STATUS_SUCCESS を返します.
//          それ以外は, システムへエラーが起きたことを示します.
//  解説:

NTSTATUS KIT1050::DriverEntry(PUNICODE_STRING RegistryPath)
{
    t << "In DriverEntry¥n";


     // このドライバの[Parameters] キーを開きます.
    KRegistryKey Params(RegistryPath, L"Parameters");
    if ( NT_SUCCESS(Params.LastError()) )
    {
#if DBG
         ULONG bBreakOnEntry = FALSE;
        // レジストリから「BreakOnEntry」値を読み取ります.
         Params.QueryValue(L"BreakOnEntry", &bBreakOnEntry);
        // 要求されている場合は, デバッガ内で停止します.
         if (bBreakOnEntry) DbgBreakPoint();
#endif
        // レジストリからドライバ データメンバをロードします.
         LoadRegistryParameters(Params);
    }
    m_Unit = 0;

    return STATUS_SUCCESS;
}


//////////////////////////////////////////////////////////////////
//  KIT1050::LoadRegistryParameters
//  ルーチン ディスクリプション:
//      レジストリからドライバ データメンバをロードします.
//  パラメータ:
//      Params - [Parameters] を指すレジストリキーをオープンします.
//  戻り値:
//      なし
//  解説:
//      メンバ変数は, レジストリから読み取られた値で更新されます.
//      パラメータは, [Parameters] キー下の値を探します.

void KIT1050::LoadRegistryParameters(KRegistryKey &Params)
{

    m_bBreakOnEntry = FALSE;
    Params.QueryValue(L"BreakOnEntry", &m_bBreakOnEntry);
    t << "m_bBreakOnEntry loaded from registry, resulting value:
[" << m_bBreakOnEntry << "]¥n";

}
// INIT セクションの終了
//////////////////////////////////////////////////////////////////
#pragma code_seg()

//////////////////////////////////////////////////////////////////
//  KIT1050::AddDevice
//  ルーチン ディスクリプション:
//      システムは, デバイスに対応するドライバを見つけたときに呼び出します.
//  パラメータ:
//      Pdo - 物理デバイス オブジェクト. これは, 物理デバイスを表すシステム
//          デバイスオブジェクトへのポインタです.
//  戻り値:
//      NTSTATUS - 成功コードまたは失敗コード
//  解説:
//      この関数は Functional Device Object, または FDO.を生成します.
//      FDO はこのドライバを可能にし, 物理デバイスのリクエストを処理します.

NTSTATUS KIT1050::AddDevice(PDEVICE_OBJECT Pdo)
{
    t << "AddDevice called¥n";

     // KIT1050Device を生成します. Kdevice のメンバ 演算子である
     // "placement" new 形式を使用することに注意してください.
     // このフォームは, システムによってアロケートされるクラスインスタンスを
     // 保存するデバイス オブジェクトのデバイス内の領域を使います.
    KIT1050Device * pDevice = new (
            static_cast<PCWSTR>(KUnitizedName(L"KIT1050Device",
                                                        m_Unit)),
            FILE_DEVICE_UNKNOWN,
            static_cast<PCWSTR>(KUnitizedName(L"KIT1050Device",
                                                        m_Unit)),
            0,
            DO_DIRECT_IO
                | DO_POWER_PAGABLE
            )
        KIT1050Device(Pdo, m_Unit);

    if (pDevice == NULL)
    {
        t << "Error creating device KIT1050Device"
            << (ULONG) m_Unit << EOL;
       return STATUS_INSUFFICIENT_RESOURCES;
    }

    NTSTATUS status = pDevice->ConstructorStatus();

    if ( !NT_SUCCESS(status) )
    {
        t << "Error constructing device KIT1050Device"
          << (ULONG) m_Unit << " status " << (ULONG) status
          << EOL;
        delete pDevice;
    }
    else
    {
        m_Unit++;

        pDevice->ReportNewDevicePowerState(PowerDeviceD0);
    }

    return status;
}
```

の違いについて，アプリケーションからドライバをオープンする方法でリスト6.1（p.177）に示します．またドライバ内の違いをリスト6.2（p.177）に示します．

そしてウイザードの最後（ステップ10/10）に「コメントを日本語にする」というチェック項目があります．ソースコードにはコメントが付記されます．英語が苦手な方はコメントを日本語にするとわかりやすくなるでしょう．

以上の設定が終了すると，最初に指定したフォルダにEXEとSYSのサブフォルダが作成され，EXEフォルダには，テスト用のアプリケーションソースが格納されます．また，SYSフォルダには，ドライバのソースが格納されます．

## 6.2 ソースファイルの概略

KIT1050ボード用ドライバのスケルトンを生成すると，次のファイルが作成されます．

- **KIT1050.cpp**
  DriverEntryの処理を行います（リスト6.3）．
- **KIT1050Device.cpp**
  IOCTRLなどのボードの制御処理を行います．機能を追加するときはこのソースに追加するか，新しいソースを追加します．
- **function.h**
  ドライバが行う処理の定義ファイルでDriverStudioが使用するので，このファイルは編集しないでください．
- **KIT1050.h**
  ドライバの定義ファイルです．
- **KIT1050Device.h**
  デバイスの定義ファイルで，定義を追加するときはこのファイルに追加します．
- **KIT1050.rc**
  ドライバの作成者やバージョン情報が入っています．ドライバをバージョンアップしたときは，このリソースファイルも変更してください．
- **KIT1050.inf**
  ドライバ登録時のINFファイルです．

## 6.3 DriverEntry部の説明

ソースコードを見るとわかると思いますが，DDKだけで作成するときに比べて，実にシンプルになっています．これは，DriverStudioが処理に必要なライブラリを提供しているので，プログラムの構造が関数呼び出しだけで済んでいるためです．DriverEntry()は，レジストリから起動時パラメータを読み出して処理を終了します．AddDevice()は，DriverEntry()終了後にシステムから呼び出されます．この関数はデバイスクラスを作成します．

## 6.4 PCIボードのリソース取得

PCIボードのリソースは，システムからIRP_MJ_PNPメッセージがきたとき，マイナ要求がIRP_MN_START_DEVICEだった場合にOnStartDevice()が呼び出されます（リスト6.4）．この関数が呼び出された時点で，PCIボードの全リソースが取得されるので，それを取り出す処理を行います．

リストのⒶでPLXのコンフィグレーションメモリのベースアドレスを取得します．PCIinf.PLXMemSize = m_MemoryRange0.Count()でメモリのサイズを取得します．PCIinf.PLXAddr = (ULONG)m_MemoryRange0.Base()でドライバ内でアクセスする際のメモリポインタを取得します．PhyAddr = m_MemoryRange0.CpuPhysicalAddress()で物理アドレスを取得します．物理アドレスは64ビットで扱われるので，PCIinf.pPLXAddr = PhyAddr.LowPartで，下位32ビットのみ取り出しています．

同様に，リストのⒷでKIT1050ボードのメモリのリソース

〔リスト6.4〕PCIボードのリソース取得

```
////////////////////////////////////////////////////////////////////
//  KIT1050Device::OnStartDevice
//  ルーチン ディスクリプション：
//       IRP_MJ_PNP - IRP_MN_START_DEVICE のハンドラ
//  パラメータ：
//       I - 現在の IRP
//  戻り値：
//       NTSTATUS - 結果コード
//  解説：
//       物理デバイスを初期化します．通常，ドライバはここで物理リソースを
//       初期化します．システムがデバイスに割り当てた指定した raw リソース
//       リストの I.AllocatedResources()，または変換されたリソースリス
//       トのI.TranslatedResources() を呼び出します．

NTSTATUS KIT1050Device::OnStartDevice(KIrp I)
{
    PHYSICAL_ADDRESS    PhyAddr;


    t << "Entering KIT1050Device::OnStartDevice¥n";

    NTSTATUS status = STATUS_SUCCESS;

    I.Information() = 0;

    // デフォルト Pnp ポリシは，既に下位デバイスで IRP をクリアにしました．
    // 物理デバイス オブジェクトを初期化します．

    // IRP から raw リソース リストを取得します．
    PCM_RESOURCE_LIST pResListRaw = I.AllocatedResources();
    // IRP から変換リソースリストを取得します．
    PCM_RESOURCE_LIST pResListTranslated =
    I.TranslatedResources();

// TODO:     以下のパラメータがハードウェアに適していることを確認します．
//
#define MAX_DMA_LENGTH 0x100000  // 0x100000 is 1 MB

    // 割り当てたリソースを使う DMA オブジェクトのデバイス ディスクリプタを
                                                              初期化します．
    DEVICE_DESCRIPTION dd;
    RtlZeroMemory(&dd, sizeof(dd));
    dd.Version = DEVICE_DESCRIPTION_VERSION;
    dd.Master = TRUE;
```

〔リスト 6.4〕PCI ボードのリソース取得（つづき）

```
    dd.ScatterGather = TRUE;
    dd.DemandMode = TRUE;
    dd.AutoInitialize = FALSE;
    dd.Dma32BitAddresses = TRUE;
    dd.IgnoreCount = FALSE;
    dd.DmaChannel = 0;
    dd.InterfaceType = PCIBus;
    dd.DmaWidth = Width32Bits;   // PCI default width
    dd.DmaSpeed = Compatible;
    dd.MaximumLength = MAX_DMA_LENGTH;

    // DMA アダプタオブジェクトを初期化します.
    m_Dma.Initialize(&dd, m_Lower.TopOfStack());

    // KPciConfiguration のインスタンスを生成すると，メモリまたは
    // I/O ポート レンジの順序を示す Base Address Register をマップできます.
    KPciConfiguration PciConfig(m_Lower.TopOfStack());

    // 各メモリマップ領域において，NT が提供するリソースを使ってメモリマップ
    // レンジを初期化します. 一度初期化されると，システム空間内の各メモリレ
    // ンジのベース仮想アドレスは Base() メンバを呼び出すことで取得できま
    // す. CPU 空間の各メモリレンジの物理アドレスは,
    // CpuPhysicalAddress() を呼び出す
    // ことで取得できます. メモリマップされたレンジにアクセスするには,
    // inb/outb のようなメンバ関数または配列エレメント演算子を使います.
    // Base Address 0 (PLX9054 のメモリレジスタのリソース取得)
    status = m_MemoryRange0.Initialize(
         pResListTranslated,
         pResListRaw,
         PciConfig.BaseAddressIndexToOrdinal(0)
         );
    if (!NT_SUCCESS(status))
    {                                                          Ⓐ
         Invalidate();
         return status;
    }
    PCIinf.PLXMemSize = m_MemoryRange0.Count();
    PCIinf.PLXAddr = (ULONG)m_MemoryRange0.Base();
    PhyAddr = m_MemoryRange0.CpuPhysicalAddress();
    PCIinf.pPLXAddr = PhyAddr.LowPart;

    // Base Address 2 (KIT1050 のメモリレジスタのリソース取得)
    status = m_MemoryRange1.Initialize(
         pResListTranslated,
         pResListRaw,
         PciConfig.BaseAddressIndexToOrdinal(2)
         );
    if (!NT_SUCCESS(status))
    {                                                          Ⓑ
         Invalidate();
         return status;
    }
    PCIinf.MemSize = m_MemoryRange1.Count();
    PCIinf.PCIMemAddr = (ULONG)m_MemoryRange1.Base();
    PhyAddr = m_MemoryRange1.CpuPhysicalAddress();
    PCIinf.pPCIMemAddr = PhyAddr.LowPart;


    // メモリマップされた各 I/O ポート領域において，NT が提供するリソースを
    // 使って I/O ポートレンジを初期化します. 一度初期化されると，inb/outb
    // のようなメンバ関数または配列エレメントを使ってポートレンジにアクセス
    // します.
    // Base Address 1 (PLX の I/O ポートレジスタのリソース取得)
    status = m_IoPortRange0.Initialize(
         pResListTranslated,
         pResListRaw,
         PciConfig.BaseAddressIndexToOrdinal(1)
         );
    if (!NT_SUCCESS(status))
    {                                                          Ⓒ
         Invalidate();
         return status;
    }
    PCIinf.PLXioSize = m_IoPortRange0.Count();
    PCIinf.PLXioAddr = (ULONG)m_IoPortRange0.Base();
    PhyAddr = m_IoPortRange0.CpuPhysicalAddress();
    PCIinf.pPLXioAddr = PhyAddr.LowPart;

    // 初期化して割り込みと接続します.
    status = m_Irq.InitializeAndConnect(
         pResListTranslated,                                   Ⓓ
         LinkTo(Isr_Irq),
         this
         );
    if (!NT_SUCCESS(status))
    {                                                          Ⓓ
         Invalidate();
         return status;
    }

    // 割り込み処理に使用する DPC をセットアップします.
    m_DpcFor_Irq.Setup(LinkTo(DpcFor_Irq), this);

// TODO: デバイスを開始するためのデバイス固有のコードを追加します.

    // 処理を完了します.

    return status;
}

////////////////////////////////////////////////////////////////
//  KIT1050Device::OnStopDevice
//  ルーチン ディスクリプション:
//       IRP_MJ_PNP - IRP_MN_STOP_DEVICE のハンドラ
//  パラメータ:
//       I - 現在の IRP
//  戻り値:
//       NTSTATUS - 結果のコード
//  解説:
//       デバイスが停止すると，システムはこのルーチンを呼び出します.
//       ドライバは，このルーチンでハードウェア リソースを解放します.
//       基本クラスは，下位デバイスに IRP を渡します.

NTSTATUS KIT1050Device::OnStopDevice(KIrp I)
{
    NTSTATUS status = STATUS_SUCCESS;

    t << "Entering KIT1050Device::OnStopDevice¥n";

    // デバイスが停止しました. システム リソースを解放します.
    Invalidate();

// TODO: デバイスを停止するためのデバイス固有のコードを追加します.

    return status;

    // 次のマクロは，Warning Level 4 でコンパイルを可能にします.
    // 関数内でこれらのパラメータを参照する場合は，このマクロを削除します.
    UNREFERENCED_PARAMETER(I);
}

////////////////////////////////////////////////////////////////
//  KIT1050Device::Invalidate
//  ルーチン ディスクリプション:
//       システム リソースの Invalidate メソッドを呼び出します.
//  パラメータ:
//       なし
//  戻り値:
//       なし
//  解説:
//       この関数は OnStopDevice, OnRemoveDevice, OnStartDevice
//       (エラー状態で) から呼び出されます.
//       これは，各リソースの Invalidate メンバ関数を呼び出し,
//       アロケートしたシステム リソースをフリーにします.
//       Invalidate 関数をリソース，または初期化されていないリソースに
//       何度呼び出しても構いません.

VOID KIT1050Device::Invalidate()
{

    // NT はこのメカニズムを提供しないので，DMA アダプタオブジェクトの
    // システムリソースを解放する必要はありません.


    // 各メモリマップ領域のシステム リソースを解放します.
    m_MemoryRange0.Invalidate();
    m_MemoryRange1.Invalidate();

    // 各 I/O ポート領域のシステム リソースを解放します.
    m_IoPortRange0.Invalidate();

    // 割り込みのシステム リソースを解放します.
    m_Irq.Invalidate();
}
```

〔リスト 6.5〕ソースコード生成直後の割り込み処理ルーチン

```
//////////////////////////////////////////////////////////////////////
//  KIT1050Device::Isr_Irq
//  ルーチン ディスクリプション：
//       IRQ Irq の 割り込みサービス ルーチン (ISR)
//  パラメータ値：
//       なし
//  戻り値：
//       BOOLEAN - 自身の割り込みの場合は True を返します.
//  解説：

BOOLEAN KIT1050Device::Isr_Irq(void)
{
// TODO:     割込みがデバイスから発生したことを確認してください.
//           次の行の「FALSE」チェックを実際の条件と置き換えてください.
    if (FALSE)
    {
        // FALSE を返し，このデバイスが割り込みを起こさなかったことを示します.
         return FALSE;
    }

// TODO:     デバイスをサービスします.
//           最小限の処理がこの ISR で行われ，多くの処理は DPC ルーチンに
//           遅延されます.
//           次のことを行ってください：
//                 ● デバイスの割り込み発生を停止します.
//                 ● タイムクリティカルな処理を実行します.
//                 ● 多くの処理を行う DPC ルーチンをスケジュールします.
//

    // 遅延プロシージャコールをスケジュール(リクエスト)します.
    // DPC ルーチンに渡す引数を設定します.
    if (!m_DpcFor_Irq.Request(NULL, NULL))
    {
// TODO:     リクエストはすでにキューの中にあります.
//           フラグのセットや他のアクションを実行できます.
    }

    // TRUE を返し，このデバイスが割り込みを起こしたことを示します.
    return TRUE;
}
```

〔リスト 6.6〕実際の割り込み処理を記述した後のリスト

```
//////////////////////////////////////////////////////////////////////
//  KIT1050Device::Isr_Irq
//  ルーチン ディスクリプション：
//       IRQ Irq の 割り込みサービス ルーチン (ISR)
//  パラメータ値：
//       なし
//  戻り値：
//       BOOLEAN - 自身の割り込みの場合は True を返します.
//  解説：

BOOLEAN KIT1050Device::Isr_Irq(void)
{
    ULONG     iStatus;
    ULONG     ICSReg;
    LONG DMAInt = -1;


// TODO:     割込みがデバイスから発生したことを確認してください.
    // 割り込み確認
    iStatus = PCIRegPointer->STATUS;
    // DMA の割り込みを確認する
    ICSReg = PCI9080RegPointer->SHARED_ICS;
    if ( (ICSReg & PCI90X0_DMA0_INTACTIVE) != 0 )
    {
         DMAInt = 0;
    }
    else
    if ( (ICSReg & PCI90X0_DMA1_INTACTIVE) != 0 )
    {
         DMAInt = 1;
    }
    if ( DMAInt >= 0 )
         (UCHAR)PCI9080RegPointer->DMA0_COMMAND_REG |=
                                              PCI90X0_DMA_CLEARINT;

    if ( ((iStatus & STS_INT) == 0) && (DMAInt < 0) )
    { // FALSE を返し，このデバイスが割り込みを起こさなかったことを示します.
        return FALSE;
    }


// TODO:     デバイスをサービスします.
//           最小限の処理がこの ISR で行われ，多くの処理は DPC ルーチンに
//           遅延されます.
//           次のことを行ってください：
//                 ● デバイスの割り込み発生を停止します.
//                 ● タイムクリティカルな処理を実行します.
//                 ● 多くの処理を行う DPC ルーチンをスケジュールします.
//
    if ( (iStatus & STS_INT) != 0 )
    {
         // 割り込みフラグ解除
         PCIRegPointer->CTRL_0 = CTRL0_ICLR;
    }

    // 遅延プロシージャコールをスケジュール(リクエスト)します.
    // DPC ルーチンに渡す引き数を設定します.
    if (!m_DpcFor_Irq.Request(NULL, NULL))
    {
// TODO:     リクエストはすでにキューの中にあります.
//           フラグのセットや他のアクションを実行できます.
    }

    // TRUE を返し，このデバイスが割り込みを起こしたことを示します.
    return TRUE;
}

//////////////////////////////////////////////////////////////////////
//  KIT1050Device::DpcFor_Irq
//  ルーチン ディスクリプション：
//       Irq の遅延プロシージャ コール (DPC)
//  パラメータ：
//       Arg1 - ユーザー定義のコンテキスト変数
//       Arg2 - ユーザー定義のコンテキスト変数
//  戻り値：
//       なし
//  解説：
//       この関数は，割り込みの2 次処理として呼ばれます.
//       高い IRQL で動作するほとんどのコードは ISR ではなくここで動作
//       するので，ほかの割り込みハンドラは動作を続けることができます.

VOID KIT1050Device::DpcFor_Irq(PVOID Arg1, PVOID Arg2)
{
// TODO:     割り込みは，主に READ/WRITE のデータ転送オペレーションの
//           最後にシグナルを出します. 次のコードは，このオペレーションに関
//           連する IRP の完了を処理するとします. また，完了する IRP がデ
//           バイスキューの現在の IRP だとします. このコードを変更，または
//           置き換えて DPC 関数を処理します.

    // 現在の Irp を参照するため KIrp オブジェクトを作成します.
// TODO:     Wizard は，すべての Irp に対して 単一のキューを作成します.
//           追加のキューを作成した場合は，この Irp 用の
//           適切なキューを選択します.
    KIrp I(m_DriverManagedQueue.CurrentIrp());

// TODO:     Status と Information フィールドを設定し，
//           成功と転送サイズを反映します.
    I.Status() = STATUS_SUCCESS;
    I.Information() = 0;

    // PnpNextIrp はこの IRP を完了し，ドライバ管理キュー内の次の
                                                  IRP 処理を開始します.
    m_DriverManagedQueue.PnpNextIrp(I);

// TODO:     デバイスの割り込みを有効にします.

    // 次のマクロは，Warning Level 4 でコンパイルを可能にします.
    // 関数内でこれらのパラメータを参照する場合は，このマクロを削除します.
    UNREFERENCED_PARAMETER(Arg1);
    UNREFERENCED_PARAMETER(Arg2);
}
```

〔リスト 6.7〕リクエストのシリアライズ

```
////////////////////////////////////////////////////////////////////
//  KIT1050Device::Read
//  ルーチン ディスクリプション:
//       IRP_MJ_READ のハンドラ
//  パラメータ:
//       I - 現在の IRP
//  戻り値:
//       NTSTATUS 結果のコード
//  コメント:
//       このルーチンは，Read リクエストを処理します.
//       デバイスが停止したり削除された場合，KPnpDevice クラスは
//       IRP フロー制限する処理をします.

NTSTATUS KIT1050Device::Read(KIrp I)
{
    t << "Entering KIT1050Device::Read, " << I << EOL;
// TODO: リクエストをチェックします. リクエストが無効な場合に TRUE を
//            返すなら，以下の行の「FALSE」チェックを置き換えます.

    // 0 バイトを読み込むなら，常に OK を返します.
    if (I.ReadSize() == 0)
    {
         I.Information() = 0;
         return I.PnpComplete(this, STATUS_SUCCESS);
    }

    // その他パラメータのチェックをするときは，ここで行います.


    // ドライバ管理キューで処理するため，IRP をキューイングします.
    // Read 関数は SerialRead で実行されます.
// TODO: Wizard は，すべての IRP に対して単一のキューを作成します.
//            追加のキューを作成した場合は，この IRP 用の適切なキューを
//            選択します.
    return m_DriverManagedQueue.QueueIrp(I);
}

////////////////////////////////////////////////////////////////////
//  KIT1050Device::SerialRead
//  ルーチン ディスクリプション:
//       シリアライズされた READ のハンドラ
//  パラメータ:
//       I - 現在の IRP
//  戻り値:
//       なし
//  解説:
//       STARTIO キューから IRP が取り出されると，このルーチンが
//       呼ばれます. 複数のリクエストが同時に処理されることはありません.
//       このルーチンはディスパッチレベルで呼ばれます.

void KIT1050Device::SerialRead(KIrp I)
{
    t << "Entering KIT1050Device::SerialRead, " << I << EOL;
    NTSTATUS status        = STATUS_SUCCESS;

    // メモリオブジェクトを宣言します.
    KMemory Mem(I.Mdl());
    // メモリオブジェクトを使い，呼び出し元バッファへのポインタを生成します.
    PUCHAR    pBuffer      = (PUCHAR) Mem.MapToSystemSpace();

    ULONG   dwTotalSize = I.ReadSize(CURRENT);
//                                                  要求された読み込みサイズ
    ULONG   dwBytesRead = 0;
//                                                      読み込んだバイト長

// TODO:    読み込みが直ちに完了した場合は，Information と Status フィー
//          ルドを設定し，この IRP を完了してキュー内の次の IRP の処理を
//          開始するために NextIrp を呼び出します.

// TODO:    データがまだ有効でない場合は，物理デバイスにリクエストを出し，
//          ハードウェアが読み取り完了を示すまでは Information, Status,
//          NextIrp の処理を延期します. これは主に，ハードウェア
//          がデータの転送を終えたあとに呼び出す DPC の中で処理されます.

// TODO:    読み取りを完了するには，デバイスから呼び出し元バッファ [pBuffer]
//          へデータを転送します. そして，転送したデータ量を通知します:

    I.Information() = dwBytesRead;

    I.Status() = status;

    // PnpNextIrp はこの IRP を完了し，ドライバ管理キュー内の
    // 次の IRP の処理を開始します.
// TODO:    Wizard は，すべての Irp 用の単一キューを生成します.
//          追加のキューを生成した場合は，
//          この Irp の適切なキューを選択します.
    m_DriverManagedQueue.PnpNextIrp(I);

}

////////////////////////////////////////////////////////////////////
//  KIT1050Device_DriverManagedQueue::StartIo
//  ルーチン ディスクリプタ:
//       IRP が，システムによってシリアライズ I/O 用のデバイスキューから
//       取り出されると， StartIo が呼ばれます.
//       StartIO は ディスパッチレベルで呼ばれます.
//  パラメータ:
//       I - キューから取り出された IRP
//  戻り値:
//       なし

VOID KIT1050Device_DriverManagedQueue::StartIo(KIrp I)
{
    t  << "Entering KIT1050Device_DriverManagedQueue StartIo, "
                                                           << I;

    // KDriverManagedQueueEx はキャンセルが不可能な状態( NULL に設定さ
    // れたキャンセル) で Irp を提供するので，システムキューやレガシークラ
    // ス KdriverManagedQueue のようにキャンセル ルーチンを初めにクリー
    // ンにすることを心配せずに処理することができます.
    // ここに異なるキャンセルルーチンを設定したり，この Irp の処理中にほか
    // のポイントで設定することもできます.

    // デバイス クラスのシリアライズされたルーチンを呼び出せるように，
    // デバイス クラスを探します. ハンドラを
    // DriverManagedQueue クラスに移動するほうが便利な場合もあります.
    KIT1050Device *pDev = (KIT1050Device *) KDevicePTR(
                                                I.DeviceObject());

    // リクエストの処理を開始します.

    // IRP 処理を分岐します.
    switch (I.MajorFunction())
    {
         case IRP_MJ_READ:
              pDev->SerialRead(I);
              break;

         case IRP_MJ_WRITE:
              pDev->SerialWrite(I);
              break;

         case IRP_MJ_DEVICE_CONTROL:
              switch (I.IoctlCode())
              {
                   default:
                     // キューイングすべきでないリクエストをキューイングし
                     // ました.( ここに到達するべきではない)
                        ASSERT(FALSE);
                        break;
              }
              break;

         default:
             // Error - 予期せぬ IRP を受け付けました.
             // NextIrp はこの IRP を完了し，キュー内の
             // 次の IRP 処理を開始します.
              ASSERT(FALSE);
              I.Status() = STATUS_INVALID_PARAMETER;
              PnpNextIrp(I);
              break;
    }
}
```

〔リスト 6.8〕IOCTRL 処理

```
//////////////////////////////////////////////////////////////////
//  KIT1050Device::DeviceControl
///  ルーチン ディスクリプション：
//      IRP_MJ_DEVICE_CONTROL のハンドラ
//  パラメータ：
//      I - 現在の IRP
//  戻り値：
//      なし
//  解説：
//      このルーチンは Device Control requests の先頭のハンドラです.
//      StartIo ルーチンを通してシリアライズ化されていても
//      いくつかの機能はすぐに実行されます.
//      デバイスが停止したり削除された場合，KPnpDevice クラスは
//      IRP フローを制限する処理をします.

NTSTATUS KIT1050Device::DeviceControl(KIrp I)
{
    NTSTATUS status;

    t << "Entering KIT1050Device::Device Control, " << I << EOL;
    switch (I.IoctlCode())
    {
         case KIT1050_IOCTL_800:
                status = KIT1050_IOCTL_800_Handler(I);
                break;

            default:
                // サポートしていない IOCTL リクエスト
                status = STATUS_INVALID_PARAMETER;
                break;
    }

    // IRP がキューイングされていた場合，あるいはドライバが指定した
    // スキームにて遅延された IOCTL ハンドラを処理する場合は，ステータス
    // 変数は STATUS_PENDING に設定されます．この場合，単にそのステータス
    // を返し IRPはあとで完了します．あるいは，IOCTL ハンドラによって返され
    // たステータスを使って IRP を完了します.
    if (status == STATUS_PENDING)
    {
        return status;
    }
    else
    {
        return I.PnpComplete(this, status);
    }
}
```

を取得します．今回は使用していませんが，PLX の I/O ポートレジスタのリソースは，リストの©で行っています．

割り込みは，リストの Ⓓ で処理ルーチンを登録します．DDK のみで作成したときと同じで，この登録を行うと自動的に割り込みが発生するので，割り込み処理ルーチンの中身を組み込むまでは，この部分はコメントにしておいてください．

リスト 6.5(p.181)は，ソースコード生成直後の割り込み処理ルーチンのスケルトンで，具体的な処理は入っていません．最低でもリスト 6.6(p.181)に示すように実際の割り込み処理を記述してから，割り込みの登録を行うようにしてください．

ドライバをアンロードする際には，システムから IRP_MN_STOP_DEVICE が要求されてきます．このときに OnStopDevice() が呼び出されるので，この中で(実際には Invalidate() 関数で)取得したリソースをすべて解放します．

## 6.5 リクエストのシリアライズ

アプリケーションから要求された処理をシリアライズするのは，ReadFile()，WriteFile()と，DeviceIoControl() のときに行います．リスト 6.7はその概要です．アプリケーションから ReadFile() の要求がドライバに渡ってくると，Read() 関数が呼び出されます．この中ではパラメータのチェックを行い，リクエストの IRP をドライバ管理のキューに入れて処理は終了します．アプリケーションに処理は戻りません．キューに IRP が登録されると，システムがキューから FIFO の手順で IRP を取り出し StartIo() を呼び出します．

この関数は，リクエストの内容により実際に処理を行う SerialRead() を呼び出します．アプリケーションに処理が戻るのは SerialRead() の処理が終了したときになります．

〔リスト 6.9〕INFファイルのバージョン記述例

```
[Version]
Class=NewDeviceClass
ClassGUID={nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}

DriverVer=1/20/2003,1.0.0.0
```

16進数の文字列が入る

## 6.6 IOCTRL 処理

アプリケーションから DeviceControl() による処理要求は，DeviceControl() に制御が渡されます．ここで使用されるコントロールコードは，DriverStudioのウイザードで定義しますが，直接ここに追加してもかまいません．また処理要求によっては，IRP をキューイングせずに直接ここで処理してもかまいません(リスト 6.8)．

## 6.7 その他

DriverStudioで作成された INF ファイルには，ドライバのバージョンが入っていません．そのままではデバイスマネージャでバージョンを確認できないことがあるので，バージョンを追加するようにしてください．リスト 6.9の例はバージョン記述のサンプルです．このように，日付とバージョン番号を指定することができます．

まるやま・はるお　ドライバ屋

# シニアエンジニアの技術草子 参拾六之段

## ◆ 年中行事

旭 征佑

● お年玉付き年賀はがき

今年は何枚ぐらい年賀状が届いただろうか…．毎年，年の瀬になると，年賀状はなんとめんどう臭い習慣なのだろうなどと思う．しかし，新しい年に年賀状を受け取って眺めていたりすると，そんな苦労などいつの間にやら，さっぱり忘れている自分に気がついて，おかしかったりする．そんな年賀状のおかげで，滅多に会えない遠方の親戚の近況がわかるし，数十年前の古い友人と今でも交信が取れている．毎年恒例の煩わしさだったりするが，実はとってもありがたい存在でもあるのが年賀状だ．

不況や虚礼廃止の流れもあって一時期減っていた年賀状だが，最近復活しているという調査結果もよく見かける．2003年の末に大手筆記用具メーカーが首都圏のビジネスマンやOLにアンケートをとったところ，出す予定の年賀状の枚数は一人平均69枚で，前年のデータから8枚も増えたという．実際，平成16年度のお年玉付き年賀はがきの発行部数は前年より5億5000万枚も増えて44億5千万枚になっている．こういったデータをもって年賀状の復活と単純に結論付ける向きもあるが，それは早計かもしれない．

というのは，増えているのはインクジェットプリンタ用のお年玉付きはがきだけだからだ．確かに平成16年度にはインクジェット用はがきは6億枚も増やしているが，そうでない普通のお年玉付きはがきは，逆に5000万枚も減らしているのだ．ちょっと古い話になるが，インクジェットプリンタ用のお年玉付き年賀はがきが早々と売り切れて郵便局に苦情が出ていたのは平成13年のことだ．翌年の平成14年には，年賀はがきの総数は7%も減らしたにもかかわらず，インクジェット用は80%も増やしていた．それ以来インクジェットプリンタ用のはがきの発行枚数は，増え続ける一方のようだ．

インクジェット用のはがきが売れることで，お年玉付き年賀はがきが売れるようになった，と見るのが実際には正しいのかもしれない．それを裏付けるかのように，パソコンショップでは，年末に年賀状作成ソフトが高く積み上げられていた．この種のソフトは毎年新しいバージョンが出て，4,000円程度で買えることが多い．安価な割に高機能で，干支などの多くのオリジナルデータを含んでおり，割安なソフトに間違いない．

書店に行くと年賀状ソフトの解説書もかなりのスペースを占める．なかには，1000円程度で売っている解説本のおまけのCDとして，年賀状作成ソフトが付いてくるものさえある．おまけといってもばかにはできない．いったいどちらが付録かわからないほど本格的だ．この手の本は各社から出ているが，わずか1，2か月の間に軒並み20万部，30万部も売れる，隠れた大ベストセラーなのだそうだ．

今年の年賀はがきの枚数が増えている背景には，こういった事情も影響しているのだろう．

● ソフトウェアパッケージの高度化，多様化

最近のパッケージは，機能がたいへん豊富で，使い勝手もよくなってきた．なぜならば，何年もかけて何度も何度もバージョンアップを繰り返し，そのたびに新機能を追加し性能をアップしてきたからだ．それだけではない．顧客のニーズや，場合によっては世情までも取り入れ，かゆいところに手が届くように進化している．これからは目的別，ターゲットとするユーザー別などに，いろいろに多様化していくのかもしれない．

逆にある機能のパッケージソフトがほしいと思って新宿や池袋の大型店に行って探すと，たいていの場合，目的のソフトを見つけることができる．それどころか，ほとんど同機能のいろいろなメーカーの製品が棚にたくさん並んでいて，選択に悩むことすらある．製造元も，聞いたこともないメーカーから，大手パソコンメーカーの製品まで多種多様だ．ソフトウェアの開発メーカーは，いったいいくつあるのか見当もつかない．ソフトウェア開発はハードウェアと違ってそれほど投資が必要ではないため，比較的誰でも参入できることがその大きな理由だろう．ハードウェアを作成しているのは，パソコンメーカーと大手の周辺機器メーカーが中心で，あまり聞いたことのないメーカーは登場することがなくなってきたことと対照的だ．

● 認められなかったソフトウェアの価値

パソコンが登場し始めたころは，一部の開発用ソフトウェアを除いてソフトはハードのおまけと考えられる傾向が強かった．パソコンの世界でソフトウェアの価値が認められるようになったのは，1985年ごろだろうか．OAという言葉が流行りだし，16ビットコンピュータで日本語を表示できるビジネスソフトが作られるようになったころだ．このころ，今は著名な製品となったワープロソフトやデータベースが，数万円という価格で

登場し，ボチボチ成功する．

こんな時代は，バージョンアップはそれほど急ではなかった．開発環境やハードの性能が圧倒的に悪く，関数やライブラリといえばコンパイラが備える標準関数ぐらいしかなかった．さらにできあがった製品をパッケージにするのにも時間を要した．媒体のコピー，マニュアルの印刷，箱詰めなどにもかなりの手作業が入っており，2か月くらいかかることが一般的だった．だから製品開発には非常にコストがかかり，期間を要したのだ．そんなわけだから，製品開発が完了したら，同じ製品でできるだけ長い期間売り，バージョンアップはできるだけ後に伸ばすのが利益を出すための鉄則だった．

結果として，いろいろと問題が発生するのだが，何度もパッチを出して急場をしのいでいた．最近の高機能パッケージが毎年低価格で販売される状況とは，雲泥の差としかいいようがない．

● 毎年のバージョンアップに弊害はないのか

最近はトラブルが起きることも増えてきているようだ．セキュリティにかかるトラブルや他のソフトウェアとの相性の問題などに収まらず，もっと基本的なバグなどの問題も多い．ある日突然，動かなくなったり，ライセンス関係のエラーが発生するようになることもある．どう見ても，基本的なテストが不足しているものもある．メーカーに問い合わせても，生半可な回答しか返ってこない．そうこうしているうちに翌年新バージョンが発売されて買い直すなどということになる．

販売店側にも混乱をきたしているのかもしれない．ある著名なソフトウェアの最新版のリリースを待ち望み，早速，最大手の電気店に行って買ってきた友人がいる．しかし，買ったのは旧バージョンだと気がついたのは箱を開けてからだった．もう一度売り場に行って確認してみると，外箱もまるでそっくりな最新版と旧版がそれぞれ5～6本ずつ棚に一緒に並べてあったそうだ．もちろん，同じ値段で販売している．店で交換を申し出たが，開封したということで応じてもらえなかったらしい．本人の不注意とはいえ，ちょっとかわいそうでもある．

ここで，メーカーの観点から勝手に考えてみよう．毎年バージョンアップすることで，低価格で商品を販売することができるから顧客を増やせる．多様化する市場にキャッチアップし，重要な障害についてもすばやく解決することができる．見方によっては，多くのユーザーを製品出荷テスト，ベータテスタに利用することもできる．




ユーザーにとっても，低価格で高機能の商品を入手できるメリットは大きい．しかし，ともすればわずかな機能の追加に惑わされ，毎年お金を払わされることにもなる．これでは高価なソフトの分割払いにすぎない．それだったら，十万円近くする高価なソフトでも，バージョンアップが遅いほうがいい．実際，対投資効果をシビアに判断される業務用ソフトウェアにおいては，確かに後者の傾向が強い．

わずかな機能の追加なら無料で行って，以前のように息の長い商品作りをしてもいいのではないか．せっかく本を処分して室内を整理した筆者だが，現在は多くのソフトウェアパッケージの外箱がパソコンの横に並んでいる．みずからの判断力の甘さに失笑する毎日だ．

あさひ・しょうすけ　テクニカルライター
イラスト　森 祐子

# Engineering Life in

## インドに流れ出るシリコンバレーエンジニアの仕事

IT バブルが弾けて早くも4年が経過したが，アメリカ全体の失業率は依然として高い．とくにシリコンバレーや北カリフォルニアでの失業率は8％に近く，全米でもっとも高い数字となっている．2003年11月にアーノルド・シュワルツネッガー氏が知事更迭選挙で勝ったのも，カリフォルニア州の財政赤字が原因とされている．ストックオプションをもらうエンジニア達は高額納税者だし，カリフォルニア州の財政のほとんどは州の所得税で賄われてきたが，IT バブル崩壊と共に多くのエンジニア達は失業者となっていった．

最近では，やっとアメリカ全体の景気が上向きはじめているが，Job loss Recovery ——つまり雇用が増えるどころか，減る景気回復と呼ばれている．これは，企業の業績が上向きはじめているものの，レイオフされた人員を補填するための雇用を行わず，人数の少ないまま仕事を続けているということだ．

シリコンバレー企業の多くは，収益を確保するためにコスト削減を行っているが，もっとも効果的なのがレイオフだ．小出しにレイオフすることによってメディアの目を避けているものの，エンジニアの雇用を減らしていることは確かだ．

そして，一般エンジニアの間ではこれによって海外に仕事が流れ出ることが懸念されている．インドや中国をはじめとして，一部は東欧，旧ソビエトなどに流れ出ているといわれている．とくに，もっともホットな場所はインドである．

### ☆ インドのメリット・デメリット

インドに仕事の一部を流すのは新しい試みではなく，シリコンバレーでは以前からもあった．シリコンバレーにはインド出身のエンジニアや経営者，そして投資家も多いことから，すでにコネクションはあった．それをうまく利用しようということだ．インドには優秀な技術者が多く，Indian Institute of Technology（IIT）といった世界的に有名な工学専門の大学もある．またインドにはプライベートなお金が集まりやすく，ベンチャーキャピタル的な投資も行いやすい環境がある．その一方で中国は比較的に投資が難しいとされている．

そしてインドは知的所有権を尊重する傾向が強いので，中国のように法的な心配が少ないという点がメリットとされている．ただ，高速道路などの交通面や，ビルなどのインフラ面では中国が整っていて，インドは遅れているとされる．

シリコンバレー企業がインドの技術力を活用するケースにはいくつかのパターンがある．まずは，インドに拠点や支社を設けて共同で仕事を進めるケース．他のケースでは，インドのアウトソース専門の会社に発注するというパターンだ．いずれも優秀な技術者がシリコンバレーの10分の1近くのコストで雇えるうえに，英語が通じるという点で大いに魅力があるようだ．

また，時差のつごうでアメリカが夜のときインドは昼なので，アメリカのエンジニア達が寝ている間にインドのエンジニア達が仕事が進め，翌日にバトンタッチできるという，“follow the sun”──「太陽を追う」開発スタイルが一時期流行りとなった．実際のところ，うまくコミュニケーションを取ったり，役割分担を決めたりしなければうまくいかないため，仕事管理のオーバヘッドが大きくなるのが現状だ．しかし，うまく回り始めるとそれなりの成果が出るのは確かなようだ．筆者の知っている会社では，回路設計をアメリカで行い，検証やレグレッションをインドで担当している．検証用のシミュレーションスイートやレグレッションのファイルを用意するのは人海戦術的で人手がかかるので，インドで行っているそうだ．また，実際の企画とスペックだけシリコンバレーで行い，実際のコーディング関係はすべてインドというスタイルもある．いずれのケースも仕様書をしっかり書いてプロジェクト管理をしっかり行っているのでうまくいっているようだ．

インドに支社や自分たちの拠点を設けていない場合は，アウトソースを利用する．このようなところは会社化されているところもあれば個人のようなところもある．いずれもシリコンバレーで仕事をしていたエンジニア達が間に入ってプロジェクト管理や仲介の作業を行うそうだ．最近のところでは，新しいベンチャーでもインドに対する考えを明確にビジネスプランで示さなければならないそうだ．インドや中国に安価なエンジニアリング資源があるわけだから，それをどれだけ新しいベンチャーが活用しているか，投資家達も知りたいわけだ．

### ☆ 他のホワイトカラーの仕事も輸出される

エンジニアリング以外の仕事もインドにシフトしている．カスタマサービスとかマニュアル類を書くテクニカルライターの仕事などが典型的なパターンだ．オンラインショップのみのDell のカスタマサービスは，アメリカのフリーダイヤルにかけてもインドの拠点に通じるという．実際，筆者もDell のパソコンを使っていて，一度お世話になったことがある．電話をすると，微妙にインドなまりが多少聞こえたが，「ジョン」と名乗るサービスオペレータが対応してくれた．なかなかていねいで文句のないサービスだった．

この経験の前後にこちらのテレビのドキュメンタリーで見たのだが，このようなアメリカ企業のカスタマサポートをインドに輸出している話だった．アメリカの顧客からサポートがインドで行われているということをわからないようにするために数々の努力やくふうを凝らしている．とくにトレーニングなどには力が入っているようだ．インドなまりを直して，アメリカ英語に教育する専門家がアメリカから派遣されるという話だ．また，逆にアメリカの地域別のなまり，俗語や言い回しを聞き取れるようにするトレーニングもあるようだ．さらにアメリカから持

H. Tony Chin

ち込んだテレビ録画のトレンディードラマやスポーツ中継，そしてニュースなどを見て，最近の一般情報，娯楽情報までをも教え込んでアメリカの客に違和感のないようにする努力をしている．テレフォンサービスを行うインド人の若者も，じつはIITの工学部卒業生だったりして，かなり高学歴なケースが多い．

インドはエンジニアの人口が多いのでトップ1%ぐらいでないと外資系のエンジニアリングの仕事についたり，海外での仕事ができないそうであるが，テレフォンサポートにしておくには少しもったいない人材が多い．その一方で，テクニカルライターだとなまりなどの心配はない．ただエンジニアから得た情報からマニュアル類を書いていくだけだ．いずれも人手がかかったり，工数が必要な作業がインドにアウトソースされるようだ．

**☆ エンジニアリング以外の仕事もインドへ**

海外に輸出される仕事はエンジニアやエンジニア関連の仕事だけではない．たとえば，バイオテックなどの基礎研究の一部をインドに持っていくケースがある．シリコンバレー近辺では電子機器やソフトウェア以外に，多少のバイオテックの企業が存在する．あまり詳しいことはわからないが，基礎研究の分野でデータ収集のために実験や分析を進めるものがあり，人手がかかるので，やはり海外の技師を使うケースが多い．

このようなケースには，前述のインド以外に台湾がある．台湾では，電子機器の製造業が中国大陸にシフトしていき，その代わりになる事業としてバイオテックに取り組んでいる．いずれも人手がかかり，アメリカでは高くつくのでまとまった人材が集中しているインドや台湾に仕事が流れる．台湾では，化学や生物学の博士号を取得した技師がアメリカの3分の1程度のコストで雇えるし，バイオテックに対する投資も盛んなので新しいベンチャー企業が立ち上がっている．アメリカのバイオテックの企業と共同開発または委託研究を進めているケースも多い．

また，新素材などの基礎研究をインドの企業に委託することがある．ソフトウェアの場合と同じで，レベルの高い技術者を安く雇えることに非常に大きな期待がかけられている．

技術職以外では，金融界のリサーチなど，バックオフィスと呼ばれている後処理的な業務がドンドン海外で行われており，インドにいくケースがもっとも多い．たとえば銀行のローン申請での書類チェックや審査などをインドの会社に委託したり，証券会社の顧客に提出するリサーチデータ類もインドの会社に委託するケースが増えている．エンジニアリングの仕事と同じように人手が必要で自動化できない仕事や工数が多い仕事に集中していることが特徴である．

**☆ 今後のシリコンバレーはどうなる？**

実際のところ，仕事がインドに移行したため，職を失ったりポジションがなくなったといった詳しい統計は，今のところデータが揃っていないらしい．データを取るのが難しいというのがその理由のようだ．会社側としてはっきり言えない立場もあるそうだ．

海外に輸出されていく職業をどう捉えるか？　これはシリコンバレーで仕事をする人々の立場によって異なる．仕事を失ったり，年齢が高かったりするエンジニア達は今後の不安を隠せず，アメリカ政府に何か政策を打ち出すように主張する人も多い．その一方でさまざまな仕事がグローバル化によって輸出されていくことを止む終えないことだと捉える人も多い．またインド系や中国系のエンジニア達は自国に仕事が流れていることを複雑な気持ちで見ているようだ．インド系のエンジニア達は，これまでインドが貧しい国であったのと経済鎖国に近いことをしてきたので，世界的に重宝される人材は国にとって良いし，今後インドの発展につながるのではないか，と見るらしい．しかし，これは最終的に地元経済が潤わなければ意味がないし，コスト的なことも単なる一時的なものであるともいえる．

またシリコンバレー全体で今後何をしていくのか？という議論も出ている．つまり，ほとんどのエンジニアリングや基礎技術の研究が国外に流れれば，何がシリコンバレーに残るのか？ということだ．今後はシリコンバレーがそれほど技術をリードしないのでは？といった悲観する意見もあれば，楽観視する意見もある．楽観している意見では，人手がかかる作業を安い場所でやれば，それだけシリコンバレーのエンジニア達が他の仕事に手が回り，他の製品開発や研究に着手できるから良いのだ…と説明する．

いろいろと賛否両論はあるし，まだまだ結果や影響が目に見えないように感じるのであるが，現在だと不況の影響もあり，少し悲壮感があるような意見が多い．しかし過去を振り返ると80年代のシリコンバレーは日本企業のパワーに圧倒され，90年代は中国の生産力に危機を感じ，今回はホワイトカラーの仕事のオフショア化に直面している．その中でシリコンバレーの内容も変わっていっているのは確かである．少なくても何かの変化や状況に対応するパワーはあるかと思う．アメリカでは一人の人間が同じ仕事を一生続けるということはほとんどなくなってきており，それが加速化しているのは間違いないと思う．エンジニア達も技術の変化やグローバル化の流れの速いところで仕事と生活を続けているのだと実感する．

トニー・チン　htchin@attglobal.net　WinHawk Consulting

●統合コミュニケーションプロセッサファミリ

## RC32434Interprise

- 最大400MHzで動作する，32ビット MIPS 4Kc CPUコアを採用したプロセッサ．
- ディジタルホームネットワークを対象に，最適な性能，柔軟性を提供．
- ×16デバイスをサポートする，DDRメモリコントローラを搭載．
- 32ビットのバージョン2.2 PCIコントローラをサポート．
- 専用のローカルメモリI/Oコントローラを搭載．
- 最大速度100MbpsのEthernetをサポート．
- 標準のMIIまたは縮小型MIIのオプションにより，広範囲なネットワーキングデバイスに対する接続が可能．

●価格：
RC32434/266MHz $15.50（10,000個時）
RC32434/300MHz $17.00（10,000個時）
RC32434/350MHz $19.00（10,000個時）
RC32434/400MHz $23.00（10,000個時）

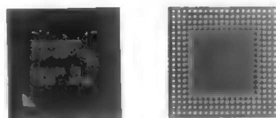
■日本IDT（株）
TEL：03-3221-6726　FAX：03-3221-5456

●32ビットRISCマイクロプロセッサ

## SH7720

- 最大動作周波数が133MHzのCPUコア「SH3-DSP」を搭載し，173MIPSの処理性能を実現．
- SSL処理の中でもソフトウェア処理の負荷が大きい3DES処理およびRAS演算用のハードウェアアクセラレータを内蔵することで，セキュアなブラウザ表示を実現．
- 0.15μmのCMOSプロセスを採用し，低消費電力化を図っている．
- カラーLCDコントローラ，USBホストとファンクション機能，コンパクトフラッシュメモリカードのインターフェースに加え，マルチメディアカード，I²Cバス，IrDA（Ver1.0）などのインターフェースを追加．

●サンプル価格：¥2,500（HD6417720BP133）
¥2,600（HD6417720BL133）



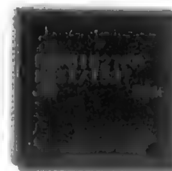
■（株）ルネサスソリューションズ
TEL：03-5201-5062

●32ビットCISCマイクロプロセッサ

## H8SX/1527F

- 車載制御システム向けCISCマイコン．
- 演算器および内部バス幅が32ビットのCPUコア「H8SX」を搭載し，最大動作周波数40MHzで，40MIPSの性能を実現．
- 命令セットは「H8Sファミリ」の命令セットに加え，新規命令の追加，アドレッシングモードの強化が行われている．
- 車内LAN規格のCANに対応したコントローラをはじめ，多機能タイマやDMAC，センサや他のシステムとの接続が容易なSSUなど，車載制御システム向けに適した汎用性の高い周辺機能をコンパクトに内蔵．
- 1サイクルアクセスが可能な256Kバイトのフラッシュメモリを搭載し，高速な処理速度を実現．

●サンプル価格：¥1,500



■（株）ルネサス テクノロジ
TEL：03-5201-5212

●マルチメディアアプリケーションプロセッサ

## OMAP1710

- 携帯電話およびモバイル機器向けのプロセッサ．
- 90nmプロセス技術を活用し，アプリケーションの性能を最大40%向上させることが可能．
- 高周波帯域への対応，データキャッシュと命令キャッシュの大容量化，および静止画像，グラフィックス，フルモーションビデオ，オーディオなどのマルチメディアアプリケーションをサポート．
- プロセッサ内のクリティカルシステムブロックの再設計によって，電力消費量を50%削減．
- 柔軟性の高いソフトウェア，マルチメディアおよびグラフィック処理性能の向上，ハードウェア/ソフトウェア双方でのセキュリティ機能の集積化，高性能カメラインターフェース，強化されたペリフェラル群および待機時の低消費電力を提供．

●価格：下記へ問い合わせ

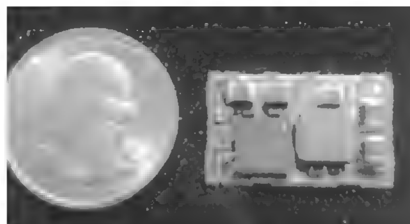
■日本テキサス・インスツルメンツ（株）
FAX：0120-81-0036
URL：http://www.tij.co.jp/pic/

●DC-DCコンバータ

## LSJ-T/7-W3

- 100A/μsの高速負荷応答性，92%の高効率の非絶縁型DC-DCコンバータ．
- 出力電圧は外付け抵抗1本で，DC1.0V～3.3Vの範囲で任意に設定することが可能．
- 入力電圧は，DC3V～5.5Vの広入力電圧範囲をサポート．
- －40℃～85℃の広使用温度範囲で，ヒートシンクの必要がない．
- 最小負荷が不要で，リモートオン/オフの制御機能付き．
- 70mVppの低リプル/ノイズを実現．
- 25×15×4mmの小型SMDパッケージで提供．

●価格：¥5,140（1～24個時）



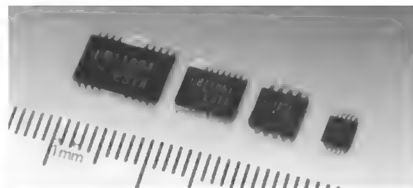
■デイテル（株）
TEL：03-3779-1031　FAX：03-3779-1030

●可変速ファンモータドライバIC

## LB11860T LB11861/M/H

- 単相バイポーラ駆動のファンモータドライバ．
- 外部信号によるダイレクトPWM制御方式を採用しており，PCメーカーが一般的に対応していたサーミスタを使用する印加電圧速度制御方式などと比較して，外部回路が大幅に簡素化される．
- LB11861/M/Hは熱分散用抵抗を付加する回路構成に対応し，ドライバICへの熱負荷が低減できるため，余裕のファンモータドライブを実現．
- 空冷ファンでの応用だけでなく，水冷システムでの水循環ポンプにも応用可能．
- 水冷ポンプにおいて重要となる可変速機能，静音化を容易に実現．

●サンプル価格：¥100




■三洋電機（株）
TEL：0276-61-8107　FAX：0276-61-8730

●PWMコントローラIC

## IRU3073

- 2種類の電圧を出力できるPWM(パルス幅変調)コントローラIC.
- DDRメモリ回路，コンピュータグラフィックスカード，デスクトップPCのマザーボードで使われるDC-DCコンバータなどの用途に適する.
- LDO(低ドロップアウト)コントローラを内蔵しているため，この出力とPWMコントローラ出力で2種類の電圧を作ることができる.
- 入力電圧は単一5Vまたは12V.
- ローサイド駆動回路とハイサイド駆動回路は独立にバイアス電圧を与えることが可能.
- スイッチング周波数の範囲は，200～400kHzで設定可能で，コンデンサを外付けして出力の立ち上がり時間を設定可能なソフトスタート機能を備える.
- DDR電圧トラッキング用に使える誤差アンプ，低電圧出力コンバータ用の0.8V基準電圧源などを内蔵.

●サンプル価格：¥180

■インターナショナル レクティファイアー ジャパン(株)
TEL：03-3983-0086　FAX：03-3983-0642

●SiGeトランジスタ

## HSG1001 HSG1002 HSG2001

- 無線LANやコードレス電話用途向けに，5.8GHz/2.4GHzなどの高周波信号の増幅を，高性能で実現するSiGeトランジスタ.
- HSG1001/HSG1002は，受信用の低雑音増幅器に使用する高周波増幅トランジスタで，周波数5.8GHzで電力利得11dBおよび雑音指数1.3dBを実現.
- 消費電力は20mWで，少ない電力で低雑音増幅器の受信感度向上を図ることができる.
- HSG2001は中電力増幅用トランジスタで，2.4GHz送信用のパワーアンプを駆動する送信用ドライバに使用可能.
- 周波数2.4GHz，電源電圧3.6Vにおいて，線形利得が12dBおよび1dB利得圧縮出力電力が+20dBmと高利得，高出力電力を実現.

●サンプル価格：¥23(HSG1001)
¥25(HSG1002)
¥25(HSG2001)



■(株)ルネサス テクノロジ
TEL：03-5201-5241

●Bluetooth半導体製品

## BlueCore3-External

- シングルチップBluetoothソリューションの中で，拡張版SCO機能に完全対応し，Bluetoothバージョン1.2仕様の機能を含むすべてのオプション機能をサポートする構成.
- フラッシュメモリを使用しており，全面的なファームウェアのアップグレードが可能となっている.
- RFユニットは，Bluetooth仕様の次世代バージョンとして提案されているPSK変調方式の送受信機能を装備.
- BlueCore2-Externalより多くの追加RAMを搭載しており，複雑なスキャッタネットを構築でき，多くのデバイスと同時接続が可能となっている.
- 大容量のデータバッファにより，次世代バージョン仕様の高速データ伝送を実現可能としている.
- 既存バージョン1.1仕様で通信する際に，Bluetoothと802.11b/gの二つの技術間の電波干渉を減少させ，二つの技術を共存させるための改善がなされている.

●価格：下記へ問い合わせ

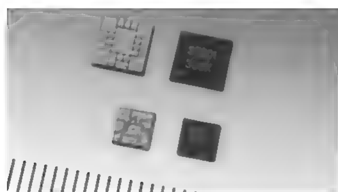
■シーエスアール(株)
TEL：03-5328-1400　FAX：03-5328-1403
E-mail：csr@e-e.co.jp

●DC-DCコンバータ電源IC

## SR10010 SR20010

- SR10010は1チャネル，SR20010は2チャネルのDC-DCコンバータコントロールIC.
- MOSFETパワートランジスタおよびショットキバリアダイオードを独自の高密度実装技術ISBを用いてワンパッケージに内蔵したSIP.
- ディスクリート部品で構成する場合と比較して周辺部品が少なく，部品占有面積はSR10010で約35%，SR20010で約42%の削減が可能.
- スイッチング電源の採用により省電力対応となっており，2次電池を使用する小型機器に最適なデバイスで，セット内の電源追加にも容易に対応できる.

●サンプル価格：¥120(SR10010)
¥200(SR20010)



■三洋電機(株)
TEL：0276-61-8253　FAX：0276-61-8958

●TFT液晶ドライバ

## HD66371 HD66372

- HD66371は420出力，HD66372は480出力のTFT液晶パネル用ドライバ.
- 回路面積の増加を抑えた新規開発の10ビットD-Aコンバータにより，256階調(1677万色表示)液晶ドライバとほぼ同等のチップサイズで，1024階調(10億7000万色表示)を実現.
- 外部から加えるガンマ基準電圧の入力方法を時分割で行うことで，RGBそれぞれを独立したガンマ設定にすることが可能.
- 液晶パネルの色調整において，ディジタルデータ処理では表現できないアナログ的なガンマ調整により，微妙な色調整が可能となり，高画質化を図ることができる.
- 液晶パネル内のRGBガンマ特性のばらつき補正なども可能.

●サンプル価格：¥900(HD66371)
¥1,000(HD66372)



■(株)ルネサス テクノロジ
TEL：03-5201-5226

●ID型共振タグシステム

## ガレット

- ID型共振タグ(磁気コイル荷札)システム.
- 高いエッチング技術により，特殊なコイルを製造し，それぞれ固有の周波数をもつ小型タグを14パターン開発.
- 異なるパターンのタグを対象物に貼り付け，独自のスキャナで読み取ることで，ID識別を行うことが可能.
- タグを複数枚組み合わせて対象物に貼ることで，最大16,383種類の対象に，個々のID識別を行うことが可能.
- ICタグの最大1/50の低コストを実現.
- 電波を利用することで，バーコード式では難しい，被覆での認識が可能.
- 物流管理，在庫管理，真贋検知などの分野に適する.

●価格：¥5～¥7(100,000個以上時)

■(株)ジーエスシーエム
TEL：03-5638-1143　FAX：03-5638-1142



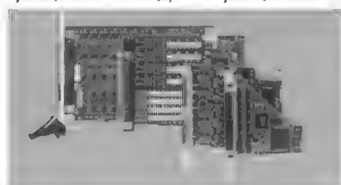
■弊誌では新製品に関するニュースリリースを募集しております.
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　(編集部)

●D-Aデータ集録デバイス

## PCI-6509/PCI-6514 PCI-6515/PCI-6722 PCI-6723

- NI-DAQmxソフトウェアテクノロジに対応しており，LabVIEW，C，Microsoft Visual Basic .NET，C#などの開発言語でアプリケーションを作成するための自動コード生成機能を搭載したD-Aデータ集録デバイス．
- NI DAQアシスタント機能を用いることにより，プログラミングを行うことなく，データの計測，テストを行うことが可能．
- NI-DAQmxではマルチスレッドストリーミングテクノロジが提供され，同時並行処理や多チャネルシステムで最大100倍速いI/Oスループットを実現し，自動タイミング，トリガ，同期などの機能を提供．

●価格：¥36,000 PCI-6509
¥42,000（PCI-6514）/¥42,000（PCI-6515）
¥101,000（PCI-6722）/¥144,000（PCI-6723）



■日本ナショナルインスツルメンツ（株）
TEL：03-5472-2970　FAX：03-5472-2977
E-mail：prjapan@ni.com

●CompactPCI規格CPUモジュール

## ACP-128-1

- 1.6GHzで動作するインテル社Pentium Mを搭載し，FSBが400MHzの省電力システムバスをもち，チップセットに855GMEとインテル社ICH4を採用．
- メインメモリはPC2100/PC2700対応のDDR-DIMMを最大2Gバイト搭載可能．
- RS-232-C×2ポート，USB×6ポート，GビットEthernet×2ポート，GPIO×1ポートなど，多彩なインターフェースを搭載．
- ボード上にUltra DMA mode2対応のIDEをもち，2.5インチHDDを搭載可能．
- PMCスロットをもち，さまざまな機能のボードをアドオンすることが可能．
- USBストレージから，OSのブートができる．
- 1Mバイトの2次キャッシュメモリを，オンダイにより搭載可能．
- PICMG2.0 Revision3.0準拠．
- +5Vおよび+3.3V電源の入力を，シーケンスフリー対応．
- サポートOSは，Windows 2000/XP，Linux．

●価格：¥288,000

■（株）アバールデータ
TEL：042-732-1030　FAX：042-732-1032
E-mail：sales@avaldata.co.jp

●T-Engine応用製品

## T-Cube（仮称）

- T-Engineプロジェクトの成果を活かした半応用製品．
- 標準T-Engineボードには含まれない，LAN機能や高解像度グラフィック機能などを追加することにより，単体で業務用端末やIAの用途に利用可能．
- 高機能，高性能な組み込み制御用のコンピュータとして幅広く利用可能．
- 回路図などの技術情報が公開されているため，ユーザー自身が最終製品向けのハードウェアを再設計し，ユビキタス機器へと展開させることが可能．
- CPUにはNECエレクトロニクス社の「$V_R$5701」を採用．

●価格：
下記へ問い合わせ



■パーソナルメディア（株）
TEL：03-5702-7858
E-mail：te-sales@personal-media.co.jp
URL：http://www.personal-media.co.jp/te/

●T-Engine開発キット

## ARM926-MB8 開発キット

- 富士通製のCPU「MB87Q1100」を搭載したT-Engine開発キット．
- 英国アーム社の開発したARM926EJ-SコアとARM946E-Sコアを搭載．
- 「MB87Q1100」は，マルチCPUの性能をフルに活かすためにバスとしてマルチレイアAHBを採用し，外部AHB拡張機能をもち，外部にFPGAなどを接続することでASICのプロトタイピングを行える特徴をもつ．
- 標準T-Engineのほか，RTOS「T-Kernel」，開発用基本ミドルウェア，GNU開発環境，仕様書などのドキュメント類が含まれる．
- Linux搭載などの開発用PCを用意するだけで，T-Engine上のミドルウェアやアプリケーションの開発が可能．

●価格：下記へ問い合わせ

■パーソナルメディア（株）
TEL：03-5702-7858
E-mail：te-sales@personal-media.co.jp
URL：http://www.personal-media.co.jp/te/

●Linuxボード

## E!Kit-1100

- 110×90mmサイズのボードに，400MHzで動作するAMD社のCPU「Alchemy Au1100」，128MバイトのRAMを搭載．
- 100Base-TX LAN，USBホストコントローラ，USBターゲットコントローラ，CF2スロット，シリアルポート，JTAG，LCDコントローラ機能を提供．
- 標準でGNUのクロスコンパイル，PC上のLinuxと同様の使い勝手のセルフコンパイル環境，独ロータバッハ社製JTAGデバッガをサポート．
- マルチメディア関連の実験や研究用途に適するソリューションを提供．
- PC上のLinux環境からの移行のしやすさと，組み込みLinuxボードとしての扱いやすさを追求．

●予価：¥98,000

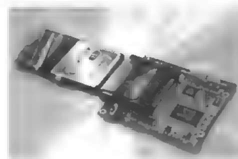
■（株）デバイスドライバーズ
TEL：042-363-8294　FAX：042-363-8255
E-mail：e-kit@devdrv.co.jp
URL：http://www.devdrv.co.jp/

●Linuxボード

## CAT709[SH7709S]

- 32ビットRISC CPUのSH-3を採用した，高性能でコンパクト，低消費電力なマイクロコンピュータボード．
- ボード上のフラッシュメモリにブートローダとSilicon Linux（シリコンリナックス社製）をインストール済みなので，標準でシェルやTELNET，FTPなどのネットワーク機能を利用することが可能．
- フラッシュメモリからLinuxが起動し，フラッシュメモリをディスクとして使用できるため，GCCで作成したアプリケーションプログラムなどを実装可能．
- ボード上のCFソケットはType IIを実装しているため，コンパクトフラッシュのほか，マイクロドライブ，無線LANカード，通信カードなどを使用できる．

●価格：¥37,500



■エーワン（株）
TEL：0568-85-8511　FAX：0568-85-8501
URL：http://www.aone.co.jp/cat/
http://www.si-linux.com/

●T-Engine向けTCP/IPパッケージ

## Nucleus NET

- 横河ディジタルコンピュータ(株)のT-Engineプラットホームに TCP/IPプロトコルパッケージ「Nucleus NET」を移植.
- 組み込みデバイス特有の要求に対応できるように,デバイスが必要としないプロトコル,あるいはプロトコル部分を簡単に除外できるスケーラブルな設計になっている.
- ソケットAPIの提供により,組み込みアプリケーションがインターネット上の他のホストと通信できる.
- 基本的なネットワーククライアントとサーバの作成方法を例証するデモンストレーションアプリケーションが付属.
- TCP,UDP,IP,ICMP,ARP,RARP,BOOTPクライアント,DNSリゾルバ,DHCPクライアント,RIP/RIP II,TFTPクライアントなどのプロトコルをサポート.

●価格:下記へ問い合わせ

**■メンター・グラフィックス・ジャパン(株)**
TEL:03-5488-3041 FAX:03-5488-3032

●ファームウェア開発支援ツール

## EZ Gear

- (株)エヌエフ回路設計ブロックが開発した,PCベースの開発支援ツール群.
- 本体は120×100mm(CDケースサイズ),重量は約200gの小型軽量タイプ.
- 「EZ1960」は,最大20kHzで12時間のパターンが作成可能なファンクションシンセサイザ.
- 「EZ1660」は,最大32チャネルで12時間のパルスパターンを出力可能なロジックジェネレータ.
- 「EZ5840」は,2チャネルで999時間の長時間記録が可能なデータロガー.
- 「EZ5850」は,24チャネルで最大24時間の長時間記録が可能なロジックアナライザ.
- USB接続で,バスパワーを使用するためAC電源が不要.
- PC上で,データの加工などが行える.
- データロガーで採取した波形をファンクションシンセサイザで再現できるなど,データ連携をサポート.

●価格:下記へ問い合わせ

**■橘テクトロン(株)**
TEL:03-3719-2261 FAX:03-3793-1329
E-mail:sales.nf@tachitec.co.jp

●ディジタルオシロスコープ用オプション

## CANバス信号解析機能

- ディジタルオシロスコープである「DL1640/DL1640L」用のオプション機能.
- CANバスに特化することで,通信開始時やエラー発生時など5種類のトリガ条件を設定可能.
- 見たい波形をピンポイントで探し出すことが可能.
- CANバス上の波形データを解析し,その結果を波形と同時に表示することができるため,トラブルシューティングが容易.
- データサーチ機能では,最大16001フレームの解析結果から,必要なフレームを高速に検索可能.
- 見たいデータをその信号波形とともに表示できるため,開発効率の向上に役立つ.

●価格:¥240,000



**■横河電機(株)**
TEL:0120-137-046 FAX:0422-52-6624

●ディジタルオシロスコープ用オプション

## I$^2$C+PCIバス信号解析機能<br>CAN+SPIバス信号解析機能<br>I$^2$C+CAN+SPIバス信号解析機能

- ディジタルオシロスコープ「DL7400シリーズ」用の,さまざまな電子機器で使用される,CAN,I$^2$C,SPIの3種類のシリアルバス信号の波形表示とプロトコル解析を実現するオプション機能.
- 一般のディジタルオシロスコープにはない,各プロトコルに必要な専用トリガ機能をもっている.
- トラブルシューティング時などに,観測したい波形を確実に捕捉することが可能.
- 各バス信号をアナログ電圧波形として捕捉,表示することで,プロトコルアナライザだけでは困難な,物理層レベルの信号評価やトラブルシューティングを行える.

●価格:I$^2$C+PCIバス信号解析機能 ¥180,000
CAN+SPIバス信号解析機能 ¥280,000
I$^2$C+CAN+SPIバス信号解析機能 ¥350,000



**■横河電機(株)**
TEL:0120-137-046 FAX:0422-52-6624

●MLCC用チェッカ

## AX-365C

- 大容量MLCCのテーピング工程検査に適する,高速,高精度な120Hz/1kHzディジタル大容量MLCC用チェッカ.
- 各レンジごとに直列,並列回路の設定が可能となっているため,測定条件に合わせて任意に選択することが可能.
- 直列等価回路への切り替えが可能になったことで,2端子測定時のプローブ部接触抵抗による誤差要因の解消を実現.
- 高速ALC回路を搭載しているため,測定電圧のドロップ現象により誤差要因となっていた大容量測定時においても,定電圧測定が可能.
- 通常の容量測定結果表示部の切り替えにより,tan δ値の表示が可能となり,高条件での検索測定が可能.

●価格:¥355,000
(2004年3月31日まで¥285,000)



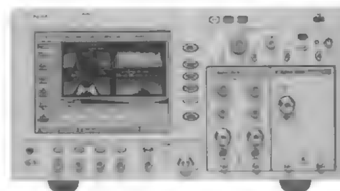
**■アデックス(株)**
TEL:075-571-2081 FAX:075-571-2089

●広帯域オシロスコープ

## 86100C Infiniium DCA-J

- アイパターン測定を行うDCA機能,ノーマライズ構成により正確なインピーダンス測定が可能なTDR機能,80GHzを超える帯域幅をもつオシロスコープ機能,光および電気信号におけるジッタ測定機能の四つの機能を1台に集約したサンプリングオシロスコープ.
- 3Gbps以上での正確なジッタ測定を実現.
- ボタン一つで,トータルジッタ,ランダムジッタ,デターミニスティックジッタ,データ依存性ジッタ,周期性ジッタ,符号間干渉,デューティ比ひずみを分離し,約11秒で数値データおよびヒストグラムの表示が可能.

●価格:約¥5,000,000



**■アジレント・テクノロジー(株)**
TEL:0120-421-345



■弊誌では新製品に関するニュースリリースを募集しております.
宛先は,〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX:(03)5395-2127,E-mail:mngnews@cqpub.co.jp (編集部)

# NEWPRODUCTS

●IP電話装置

## Applico SIP RTC スイッチ ASA880

- 同時セッション数や冗長化機能，拡張性などをミドルレンジ向けの仕様としたモデルで，一般企業を対象としている．
- G711，G723，G723.1，G729など，さまざまなコーデックに対応．
- RTCによるVoice，Video，インスタントメッセージング，アプリケーションシェアリングなどの通信を実現．
- Windows Messengerとの互換性をもつ．
- 5060ポートのダイナミックコントロールやTLS，PPTP，VPNなどの暗号化機能をサポートするなど，安全な通信を実現．
- IP-PBX，PSTNゲートウェイ，SIPサーバ，SIPクライアントなど他のコンポーネントとの相互接続を実現．

●価格：¥2,800,000～



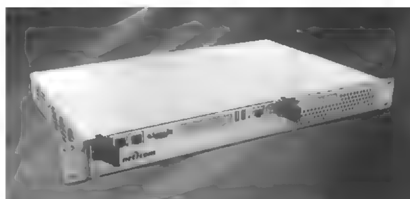
■（株）アズジェント
TEL：03-5643-2561　FAX：03-5643-2571
E-mail：info@asgent.co.jp

●クロスコネクトスイッチ

## SHOUT$_{IP}$/SCS

- 企業向けVoIP用のIPソフトスイッチ兼ゲートウェイ機能をもった，クロスコネクトスイッチ．
- 企業内線をUA（ユーザーエージェント）として認識し，装置上のレジスタで管理を行い，上位のIP事業者に対して通信時のレジストレーションを行う形式を採用．
- 一台のゲートウェイ装置で，複数の内線（最大10,000エントリ）を扱えるようにしている．
- グローバルIPアドレスの節約，クロスコネクト機能による内線交換，内蔵のSIPプロキシサーバを利用したSIPネットワークの構成が可能となる．

●価格：¥2,200,000～



■ネットドットコムジャパン（株）
TEL：03-5439-5295

●エミュレーションシステム

## CHIPitファミリ

- 複数のFPGAを利用して大規模ディジタル回路設計データを，プロトタイプとして実現するためのハードウェアと専用のソフトウェアで構成される．
- 高速動作と設計内部の状態値の観測や制御を可能にすることで，SoC設計の検証効率やソフトウェアの開発効率を向上させ，コストの削減を図ることができる．
- ザイリンクス社のVirtex II 8000 FPGAを最大18個搭載可能なアーキテクチャであり，約1000万ゲート規模の設計に対応可能．
- ボード上のFPGA間のシステム動作スピードは200MHzで，約150万ASICゲートを超える場合でも，ボード間のシステム動作スピードは100MHzのパフォーマンスを確保．
- 独自のUMRバスコミュニケーションシステムで，内部状態の観測や検証結果の視認観測が可能．

●価格：
¥37,000,000～（約300万ゲート対応構成）
¥85,000,000～（約1000万ゲート対応構成）

■（株）エッチ・ディー・ラボ
TEL：045-477-4315　FAX：045-477-4316

●タブレットPC

## FMV-STYLISTIC TB10/B

- PCの機能に手書き入力/操作機能を加えたタブレットPC．
- 従来モデルの最薄部20.9mm，横幅220mmを維持しながら，12.1型の大画面液晶（XGA）を採用．
- 独自技術によるLCD表面と保護用アクリル板との距離の最適化を図り，視差の極小化とLCD保護の両立を実現．
- 高性能で省電力CPU「超低電圧版インテルPentium Mプロセッサ1GHz」を搭載．
- チップセット「Intel 855GME」および約2.7Gバイト/sのデータ転送速度に対応したDDR SDRAMをサポート．
- 大容量バッテリ（別売）により，約7.5時間のバッテリライフを実現．

●価格：¥276,000



■富士通（株）
TEL：0120-950-222

●電流プローブ

## SS-250

- オシロスコープ用電流プローブとしては，広帯域DC～100MHzを実現し，高効率スイッチング電源の立ち上がりの速い信号を，より忠実に観測できる．
- 30A$_{rms}$（同社比2倍）の電流測定範囲を持ち，パワーデバイスなどの特性観測を余裕をもって行うことができる．
- mAオーダの観測が容易にできる，ローノイズ特性．
- 250MHz以上のオシロスコープには，プローブ用電源が用意されている．
- ノイズは，2.5mA$_{rms}$以下．
- 最大定格電力は，5.3VA．
- 定格電源電圧は，±12V±0.5V．

●価格：¥280,000



■岩通計測（株）
TEL：03-5370-5474　FAX：03-5370-5492
E-mail：info-tme@iwatsu.co.jp

●コンパイラ

## インテルC++/Fortranコンパイラ

- コンパイラの最適化機能は，開発コード名Prescottのストリーミング SIMD拡張命令3，インテルItanium2プロセッサのソフトウェアパイプライン化をサポート．
- OpenMP2.0に対応しており，自動並列化機能を通じてスレッドアプリケーションの開発と最適化をサポート．
- Windows版インテルVisual Fortranコンパイラ 8.0は，Compaq Visual Fortranのフロントエンドが Fortranのバックエンドに統合されている．Microsoft Visual Studio .NET開発環境への統合をサポート．
- Windows版インテルC++コンパイラ 8.0には，PDA向けの最適化コンパイラ，C++コンパイラ eMbedded Visual C++版が含まれる．Microsoft Visual C++ 6.0および.NETとソース/オブジェクトコードレベルで高い互換性を実現．Microsoft Visual Studio環境へのプラグイン機能をサポート．

●価格：¥65,000 Windows版Visual Fortranコンパイラ 8.0/¥89,000 Linux版/¥55,000 Windows版C++/¥55,000 Linux版C++

■エクセルソフト（株）
TEL：03-5440-7875　FAX：03-5440-7876
E-mail：intel@xlsoft.com

●PDA向けブラウザ

## NetFront v3.1 for Pocket PC

- インターネット上のさまざまなサイトを，PDAの画面サイズに合わせて見やすく再レイアウトできる．
- 横スクロールバーの操作なしで一般サイトを閲覧できる．
- 外部メモリへのインストールが可能．
- オートクルーズ機能に階層の指定，および定刻の自動起動をサポート．
- Google，Yahoo!，infoseek用のインターネット検索バーを搭載．
- PocketPCのActiveXを利用し，Macromedia FlashやWindows Media PlayerをWebページ内で再生可能．
- Personal Java実行環境JV-Lite2がXScale CPUに対応したことにより，Pocket PC2002/2003機種でJavaアプレットの動作が可能．
- Pocket PC Phone Editionをサポート（英語版のみ）．
- タブにより五つまでのウィンドウの切り替え表示ができる．

●価格：¥2,480～¥2,980

■（株）ACCESS
TEL：03-5259-3685　FAX：03-5259-3684
E-mail：prinfo@access.co.jp

●Bluetoothプロトコルスタック

## B-Rappore

- 近距離での機器の接続に特化した，無線通信技術Bluetoothプロトコルスタック製品．
- SDKを付属しており，Bluetoothを使用するアプリケーションを簡単に作成することが可能．
- MontaVista Linuxの評価が可能なソフトウェアパッケージ「Embedded Linux Reference Kit for MontaVista（ELRK）」が付加サービスとして加えられる予定．
- PDA，携帯電話，プリンタ，スキャナ，ディジタルカメラなどの製品間のワイヤレス通信製品の開発に適する．

●価格：下記へ問い合わせ

■（株）イーエルティ
TEL：03-5251-4350
E-mail：news@emblit.co.jp

●開発キット

## WirelessUSB LS開発キット

- WirelessUSB LSは，通信専用デバイス（CYWUSB6932）とトランシーババージョン（CYWUSB6934）を組み合わせることにより，ワイヤレス技術における性能と価格の最適な組み合わせを提供している．
- デバイスの通信距離は最大10mで，平均デッドサイクルは4ms未満，通信速度は62.5kbpsとなっている．
- 単一チップに高集積無線トランシーバとディジタルバースバンドを採用することで，操作範囲，電力消費およびデッドサイクルを向上させながら開発時間，部品点数およびシステムコストの大幅な削減が可能．
- 2.4GHzグローバルISMバンドを使用しているため，地域による周波数要件に関わらずソリューション展開が可能．
- 開発キットは，二つの試作プラットホームボードに直接接続される無線モジュール二つと，広範囲なWirelessUSBプロトコルコード例，関連回路図，ガーバファイル，部品表が付いたWirelessUSB「リスナー」ツールから構成．

●価格：¥90,000

■日本サイプレス社
TEL：03-5371-1921　FAX：03-5371-1955

●プリントミドルウェア

## Mobile PictDirect

- パソコンを介さずにディジタル画像を印刷するためにCIPAが策定した規格，PictBridgeに準拠した携帯電話専用ミドルウェア．
- カメラ付き携帯電話に搭載することで，携帯電話側からの簡単な操作で，画像を直接印刷することが可能．
- 印刷に使用するプリンタは，PictBridgeに対応していれば，メーカーや機種を問わない．
- Java APIをサポートしているため，ディジタル画像を印刷するだけではなく，画像処理や年賀状，シール作成などのソフトをJavaアプリケーションとして携帯電話にダウンロードすることが可能．
- ポーティングのためのアプリケーション作成用サンプルソースコードとドキュメントをあわせて提供．
- プリンタなどの画像入力デバイスおよびディジタルカメラなどの画像出力デバイスに対応．

●価格：下記へ問い合わせ

■（株）アプリックス
TEL：03-5286-8438
E-mail：pr-team@aplix.co.jp

●組み込み向けブラウザ

## T-Engine対応Esprit

- HTML，C-HTML，XHTMLなどの記述言語に対応した有線/無線環境における携帯情報機器向けの組み込みブラウザ．
- ソースコードパッケージとして提供することを前提として開発されているため，移植およびカスタマイズの自由度が高い設計になっている．
- 小メモリ，高機能をハイレベルで実現．
- W3C標準仕様を搭載．
- 各種JavaVMに対応可能．
- PDA向けJavaScriptエンジンを搭載．
- RSA社正式ライセンスのSSLの組み込みが可能．
- 対応画像フォーマットは，GIF，アニメーションGIF，JPEG，PNG，BMP．
- 文字エンコードは，JIS，シフトJIS，EUCおよびUTF8．
- 販売は，パーソナルメディア（株）が行う．

●価格：下記へ問い合わせ

■ウェブソフト・インターナショナル（株）
TEL：03-3555-3771　FAX：03-3555-3772
E-mail：esprit@wsoft.co.jp

●要件管理ツール

## DOORS/Analyst, TAU/Architect 2.2, TAU/Developer 2.2

- Telelogic DOORSの機能拡張版で，UML2.0のグラフィカルな図を使用してDOORSのデータベース内で要件をビジュアル化することを可能にする．
- 要件管理ツール内でビジュアルモデルの使用ができる機能を提供しており，理解の向上，コミュニケーションの簡素化，設計プロセスの促進をもたらす．
- TAU/ArchitectおよびTAU/Developer バージョン2.2は，UML2.0をベースとしたモデル駆動型アーキテクチャ手法によって，開発全体をサポート．
- TAU/Architectを使用すると，より効果的に大規模で複雑なシステム設計のモデリングが可能．
- TAU/Developerにより，TAU/Developer内やTAU/Architectからインポートされた検証済みモデルをベースとする，リアルタイムアプリケーションの高品質なコードの自動生成が可能となる．

●価格：下記へ問い合わせ

■日本テレロジック（株）
TEL：03-6402-1620　FAX：03-6402-1621



■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

# IPパケットの隙間から

## いまだに使われているUUCPとSPAMの被害

祐安 重夫

今ではインターネットといえば，IP 接続されたネットワークが当り前となっている．そのこと自体は昔から変わっていないのだが，筆者が，というより日本で，ユーザー名+@+ドメイン名といったメールアドレスが使用できるようになった1980年代中ごろには，ネットワークの接続とメールの配送にはIPではなく，UUCPが使用されるのが一般的だった．

UUCP(UNIX to UNIX File Copy)のしくみは単純で，送信したいデータとそのデータを処理するコマンドを組にして送信し，受信側で実際に指定されたコマンドでデータを処理してもらうというだけのものである．たとえばメールなら，本文(＋ヘッダ)をデータとして送信し，それをrmailというコマンドで処理するように指定すれば，他のコンピュータにメールを送信できた．

送信されるデータ(とコマンド)はパケット化され，モデムを経由して電話回線で通信が行われた．このままでは送受信を行うすべてのサイトの間で，直接の通信経路を確保しなければならないように思えるが，UUCPシステムやメールの配送を実際に処理するsendmailなどのMTAの設定で，自分あてのものではないパケットはさらに別のサイトに転送するというバケツリレーの原理によって，間接的にでも接続されているかぎり，どこまででも転送することができた．

UUCP自体は1976年にベル研究所で開発され，UNIXとともにリリースされるようになった．この機能を利用した最初の広域ネットワークの試みが，1979年にアメリカでデューク大学とノースカロライナ大学の間の接続から始まったUSENETである．これの日本版として1984年に始まったのがJUNETであり，日本でも個人や企業を含む広域ネットワークへの接続が，少なくともメールやネットニュースに限定されていたが，可能になった．

1980年代になり，現在のインターネットの原型ともいえるArpanetが他のIPネットワークとの相互接続を開始したり，USENETもそれに接続され，さらにネットニュースのIP転送のためのNNTP(Network News Transfer Protocol)が開発されるなど，世界規模のコミュニケーションネットワークの基盤ができあがった．

1980年代の終わりから1990年代の始めにかけて，インターネットに関するさまざまな標準や規格が整備され，インターネットの商用化が進み始める．1987年に世界最初の商用UUCP接続サービスとしてUUNETがスタートし，1992年には日本にも商用インターネットプロバイダが出現した．もっとも日本でも，当初はサービスのかなり大きな部分をIP接続ではなくUUCP接続が占めていた．

このような流れの中でネットワークへの接続が，学術ネットワーク以外は商用プロバイダ経由が原則となり，筆者のところでも二つのドメインを相継いで商用プロバイダに接続するようになったのだが，最初はUUCP接続だった．現在はUUCP接続を受け付けているプロバイダはほとんどなくなったが，実はまだ一つのドメインだけ，メールの受信(それも滅多にこない)しかしないという環境のため，もう10年ほどUUCP接続の契約になったままである．

UUCP over IP ── UUCPパケットをIPパケットでカプセル化する技術さえ使用せずに(そのプロバイダ経由のIP接続でないと，それができないのだ)，いまだに電話回線とモデムで週に3回だけ接続している．何も転送するものがなければ電話の接続時間は30秒にもならないので，KDDが1円電話を始めたときにはいち早くそれを使用し始め，現在でもKDDIから数か月に1回請求書が送られてくる．以前は毎月，500円にも満たない請求書が来ていたのだが，最近はそれでは採算がとれないということで，ある程度金額がまとまるごとに送られて来るようになったのだ．

問題は最近になって，このUUCP接続に異変が起こったことだ．突然，1回の接続でメールが数本から数十本送られて来るようになり，それがどれも存在しないユーザー宛のものなのだ．ユーザー名をほぼ届いた順に並べてみると，Akahito，Akihito，Akane，Aki，Akioといった感じで始まり，最近ではDaisetsu，Eisuke，Eitoku，Eizan，Fujio，Fujimaro，Hiroko，Hiromi，Hitomi，Hitomoといったあたりまで来ている．

これで想像がつくように，それらしい名前(日本人の名前としてはちょっと変なものも含まれているのが，御愛敬である)をアルファベット順に並べているのは，ほぼ間違いない．そういえば最近，墓地やマンションのセールスの電話の中に相手が誰かわからずにかけているとおぼしきものがかなりあり，追求すると販売対象の近辺の電話の局番について，0000から9999まで順番にかけていることを白状する輩がいたが，これと似たようなものだろうか．

筆者のところでは現在三つのドメインを保持しているが，一つはプロバイダ側にメールサーバがあるので，そこに登録していないユーザー名はプロバイダで処理される．自前でメールサーバを持っている別のドメインでは，SMTP接続の時点で排除されてしまう．ところがUUCPはメールのヘッダを見ることなく，そのまま転送するので，rmailからMTUが呼び出され処理される時点でエラーが発覚し，発信人にエラーメールが送られる．これが一つ目の問題である．

(この項つづく)

すけやす・しげお


# INFORMATION
# 海外・国内イベント/セミナー情報


## 海外イベント

2/12-15 **PMA 2004**
Las Vegas Convention Center, Las Vegas, NV, USA
Photo Marketing Association International
http://pma2004.pmai.org/

2/14-19 **ISCC(IEEE International Solid-State Circuits Conference)**
San Francisco Marriott Hotel, San Francisco, CA, USA
IEEE Solid-State Circuits Society
http://www.isscc.org/isscc/

2/16-20 **DATE04(Design, Automation & Test in Europe Conference & Exhibition)**
CNIT La Defense, Paris, France
The European Design and Automation Association
http://www.date-conference.com/

2/17-19 **IDF Spring 2004(Intel Developer Forum Spring 2004)**
Moscone Center, San Francisco, CA, USA
Intel
http://www.intel.com/idf/us/spr2004/

2/18-20 **SEMICON Korea 2004**
COEX(Convention & Exhibition Centre), Seoul, Korea
SEMI Korea
http://events.semi.org/semiconkorea/V40/index.cvn

2/23-26 **3GSM World Congress 2004**
Palais Des Festivals, Cannes, France
IBC Telecoms & Media Group
http://www.3gsmworldcongress.com/congress/

2/24-26 **IPC Printed Circuits EXPO**
Anaheim Convention Center, Anaheim, CA, USA
IPC Association
http://www.ipcprintedcircuitexpo.org/

## 国内イベント

2/1-3 第2回NIMS国際コンファレンス
湘南国際村センター(神奈川県三浦郡葉山町)
独立行政法人 物質・科学研究機構
http://www.nims.go.jp/nimsic/

2/4-6 NET&COM 2004
日本コンベンションセンター(幕張メッセ，千葉県千葉市)
日経BP社
http://expo.nikkeibp.co.jp/netcom/

2/5-6 ISS(SEMI Industry Strategy Symposium)Japan 2004
パンパシフィックホテル横浜(神奈川県横浜市)
(社)溶接学会
http://www.semi.org/wps/portal

2/18-20 PHOTONFAIR2004
東京国際フォーラム(東京都千代田区)
浜松ホトニクス(株)
http://www.hpk.co.jp/Jpn/pf/info.htm

2/12-14, 19-21 ENEX2004
東京国際展示場(東京ビッグサイト，東京都江東区)，2/12-14開催
インテックス大阪(大阪府大阪市住之江区)，2/19-21開催
(財)省エネルギーセンター
http://www.eccj.or.jp/enex2004/guide/

3/2-5 IC CARD WORLD 2004/SECURITY SHOW 2004
東京国際展示場(東京ビッグサイト，東京都江東区)
日本経済新聞社
http://www.shopbiz.jp/pages/t_index.phtml?PID=0003&TCD=IC
http://www.shopbiz.jp/pages/t_index.phtml?PID=0003&TCD=SS

4/7-9 EDEX2004(第19回 電子ディスプレイ展)
東京国際展示場(東京ビッグサイト，東京都江東区)
日本エレクトロニクスショー協会
http://www.jesa.or.jp/jp/exhibitions/edexess/index.html

## セミナー情報

ソフトウェア開発における要求の仕様化と管理法
開催日時　：2月2日(月)
開催場所　：オームビル(東京都千代田区)
受講料　　：52,500円/1口(1口で1社3名まで受講可)
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c040202n.htm

デバッグ工学　組み込みソフトウェアのシステム設計とテスト技法
開催日時　：2月2日(月)～2月3日(火)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：76,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/24/24_023.htm

モトローラ　i.MXアプリケーションプロセッサと
イーエルティ組込みLinuxミドルウェアソリューションセミナ
開催日時　：2月3日(火)
開催場所　：東京中小企業投資育成ビル8Fホール(東京都渋谷区)
受講料　　：無料
問い合わせ先：(株)イーエルティ，☎(03)5251-4350
http://www.emblit.co.jp/event.html

オブジェクト指向技術によるプロセス改善の実践
開催日時　：2月5日(木)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

DC-DCコンバータ設計の基礎
開催日時　：2月7日(土)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

JPEG2000，MotionJPEG2000徹底解説
開催日時　：2月13日(金)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

XPortソリューションセミナー
開催日時　：2月18日(火)，20日(木)
開催場所　：日新システムズ京都本社(2月18日開催，京都府京都市下京区)，
日新電機東京支社(2月20日開催，東京都千代田区，秋葉原)
受講料　　：55,000円(「X-Port Evaluation Kits」1セット付き)
問い合わせ先：(株)日新システムズ，<関西地区>☎(075)344-7881，FAX(075)344-7887，<関東地区>☎(03)5807-5931，FAX(03)3839-0112
http://www.co-nss.co.jp/event/event.html#xport

ハードディスク装置の制御事例としくみ
開催日時　：2月19日(木)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

～これからUMLを学ぶ方のための～
UML入門解説とユースケース図の書き方演習講座
開催日時　：2月19日(水)～2月20日(木)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：76,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/24/24_077.htm

CCDイメージセンサの基礎と応用
開催日時　：2月20日(金)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

CMOSイメージセンサの基礎と応用
開催日時　：2月21日(土)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

誤り訂正符号の基礎と実際
開催日時　：2月26日(木)
開催場所　：CQ出版セミナールーム(東京都豊島区)
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

～大規模ソフトベンダーにみる～ソフトウェアテスト技術と品質保証の実際
開催日時　：2月27日(金)
開催場所　：中央大学駿河台記念館(東京都千代田区)
受講料　　：52,700円
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/040223n.htm

開催日，イベント名，開催地，問い合わせ先の順
日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

# 読者の広場

## Interface への声



### 2004年1月号特集「基礎からわかるPCI & PCI-X活用技法」に関して

▷昨年，組み込みシステムでPCIバスを採用したのですが，やはり頭を悩ませたのがPCI BIOSの実装でした．CPUが知らないターゲットデバイスを実装されても見て見ぬふりをさせています．もちろんPCI-PCIバスブリッジも考慮していません．試行錯誤しながらPCI BIOSを作成しましたが，第5章を読み，やはり考え方は一緒なのだと思いました．（白石 隆）

▷バスが高速なほど良いのはわかる．PCIがPCI-Xへと高速化し，次はPCI-Expressへと進んでいる．ディジタル映像のようにそれなりに処理が重いものがバスとCPUの処理時間を消費してしまうのは確かだ．しかし，それは本来CPUではなく専用プロセッサの仕事ではないだろうか？（伊藤 啓）

［編］PCは用途によっていろいろな拡張ボードを接続したり，処理プログラムを走らせるので，結果的に汎用拡張バスを強化するという方向に進んでいるようです．たとえばその昔，PCのバス性能が低かったころはPC上で動画映像を表示させるために，動画転送専用バスとしてフューチャーコネクタが装備されたVGAカードがありました．しかし現在のビデオキャプチャカードで，フューチャーコネクタを使うものはありません．HDDやDVD上の映像も，ビデオキャプチャカードからの映像も同様に扱うには，汎用の拡張バスを使うしかないからです．組み込み機器のように用途が決まったものであるなら，専用プロセッサ＋専用バスという構成もありうるでしょう．

### Interface全般に関して

▷毎号興味深く読ませてもらっています．なかなか時間がとれず最新号まで読み進めないのですが，今後も最新技術などの解説を楽しみにしています．（KN）

▷「なぜエンジニア達は太る？」ですが，あの原因はアメリカだけの現象では？ 同じ太るにしても，各国それぞれに別の原因があると思います．エンジニアとしての共通項をもっと掘り下げてほしいですね．（原 泰已）

▷Windows XP Embeddedの記事はあまり他誌には見られない貴重な情報でした．中でウィルス/ワーム対策という記事がありましたが，もう少し詳しい解説があると実践的な内容になったと考えます．（JR9JUK）

## アンケートの結果

### 興味のあった記事（2004年1月号で実施）

①第2章　PCI-Xバスプロトコルの詳細
②プロローグ　PCIバスを取り巻く現状
③第1章　バスとは何か――PCIの基礎知識
④第3章　FPGAによるPCI-X対応デバイス設計事例
⑤第5章　PCIバスツリー構造とPCI BIOSの動作

## 特集担当デスクから

☆今回の特集は機器設計25年のベテランN先輩とA君，B君，C君という初級エンジニアとの掛け合いで構成しました．いかがだったでしょうか．会話調で話が進むとかなり冗長になりますが，そのぶんテクニカルタームが浮き彫りになって，ストレートにポイントが伝わります．まわりに先生役の先輩がいない環境の方にはわかりやすくまとまったのではないでしょうか．関数の作り方など，筆者の方も反省しているように…「この原稿を書くにあたって，自分の場合を振り返ってみると，筆者が今まで作成してきた関数は，うまくない関数のほうが多かったように思います．そこで，自分自身の反省の意を込めてCプログラムの関数について改めて考察してみました」，これで完璧というものはないようです．

☆産業用の組み込み機器は十年や二十年使われるのが普通です．二十年といえば，パソコンの代替わりは何世代も進んでいます．MS-DOS時代の「インターフェース」誌には，PC-9801シリーズ（NEC），IBM PC/AT互換機，J-3100シリーズ（東芝），AXマシン，FMR（富士通）などの広告があります．そのころのパソコンはOA用はもちろんFA用途にもかなり使われていました．OA用はつぎつぎと世代交代が進み，古いパソコンは廃棄されていきましたが，FA用は息長く使われ続けています．最近さすがに耐用年数が過ぎ，いうことを聞かなくなった旧世代のパソコンが多くなり，リプレースの時期を迎えているのですが，新しいパソコンを導入しただけでは済まず，ソフトも書き換えなければなりません．見た目は同じ仕様のソフトに大金を掛けられないのが，いまの製造現場です．また，処理系のC言語も代替わりが激しく，同じソースでもコンパイル結果が異なります．

☆FAでよく使われていたPC-9801シリーズには98アーキテクチャ互換産業用コンピュータというものが製品化されています．Cバスも搭載されているので，拡張基板も使えます．数の論理は強くて，互換機はいちばん台数の出た機械が対象になります．

☆これから組み込みソフトを開発するみなさんも考えておかなくてはならない問題ではないでしょうか．

# 読者の広場

⑥Appendix 2　組み込み機器におけるPCIバスの実装方法
⑦「IrFront H8S Trial Kit」の詳細
⑧シニアエンジニアの技術草子(参拾四之段)
⑨第4章　バスマスタPCIデバイス対応Windowsデバイスドライバ開発事例
⑩組み込みLinuxをとりまく世界(第4回・最終回)
⑪高性能圧縮ツールbsrcの理論と実装(後編)
⑫Appendix 1　ロジアナ波形でみるPCI＆PCI-Xバスの動き
⑬初級ドライバ開発者のためのWindowsデバイスドライバ開発テクニック(第4回)
⑭フリーソフトウェア徹底活用講座(第13回)
⑮移り気な情報工学(第36回)
⑯Engineering Life in Silicon Valley
⑰ハッカーの常識的見聞録(第37回)
⑱第6章　PCI拡張ROMプログラムの開発
⑲組み込みGUI設計の現状とソリューション(第2回)
⑳「VxWORKS」を使ったRTOS技術の基礎と応用(第3回)
㉑InterGiga No.32
㉒Appendix 3　PCIデバッグライブラリ for DOS新バージョン登場!
㉓CETEC JAPAN 2003
㉔開発環境探訪(第24回)

## 特集『基礎からわかるPCI＆PCI-X活用技法』についてのアンケートの結果

Q1　PCI-Xを搭載したPC/AT互換機やサーバ/ワークステーションなどを使われていますか?
①使っている(6%)
②使っていない(94%)

Q2　32ビット/33MHzのPCIでは帯域が足りないと感じたことはありますか?
①足りない(37%)
②今のところ十分(63%)

Q3　PCIデバイス/アドインカードを設計されたことがありますか?
①ある(18%)
②ない(82%)

(Q3で「ある」をお答えいただいた方に質問です)
Q4　転送レートはどの程度の性能のデバイス/カードですか?
①ターゲット仕様/数Mバイト/秒程度(50%)
②ターゲット仕様/数十Mバイト/秒程度(25%)
③バスマスタ仕様/10Mバイト/秒未満(0%)
④バスマスタ仕様/10M～50Mバイト/秒程度(25%)
⑤バスマスタ仕様/50M～100Mバイト/秒程度(0%)
⑥バスマスタ仕様/100Mバイト/秒以上(0%)

Q5　今後，PCIおよびPCI-Express関連で取り上げて欲しい内容があれば，教えてください．
- 組み込みPCIシステムの応用事例
- PCI-Express
- Linux用PCIデバイスドライバ開発事例
- デバイスドライバの最適化

## Interface 年間予約購読のお知らせ

Interfaceを確実にお手元にお届けする年間予約購読をご利用ください．

Interface：毎月25日発売
年間予約購読料金：10,800円
※予約購読料金の中には年間の定価合計金額および送料荷造り費用が含まれます．

● 申し込み方法
お申し込みは，FAXで下記までご通知ください．お申し込みに便利な「年間予約購読申込書」をWeb上でも公開しています(http://www.cqpub.co.jp/hanbai/nenkan/nenkan.htm)．こちらもご利用ください．
お支払い方法は，クレジットカード・現金書留・郵便振替・銀行振込がご利用になれます．
お申し込み受け付け後，請求書を発送いたします．

● 年間予約購読の申し込み先
CQ出版株式会社 販売局 販売部
TEL：03-5395-2141　FAX：03-5395-2106



## 読者プレゼント



●応募方法：本誌読者アンケートはがきに必要事項を記入のうえ，2004年2月29日(必着)までにご応募ください．なお当選者の発表は発送をもってかえさせていただきます．

(1) カードホルダ付き携帯ストラップ　(10名)
(株)アクセル　http://www.axell.co.jp/
(長さ約50cm)



(2) 修正テープ　(10名)
(株)アクセル　http://www.axell.co.jp/



Interface

次号予告

2004年4月号は
2月25日発売です

Ethernetの物理層からドライバ/プロトコルスタックの移植まで

# 作りながら学ぶ Ethernet活用技法

## 10Base-T／マンチェスタ符号／CSMA/CD／コリジョン検出／Ethernetフレーム

PCはすでにネットワークに接続できなければ使いものにならない道具となっている．また最近普及し始めているHDD/DVDビデオレコーダはEthernetにつながり，ネットワーク経由で予約録画などが可能になっている．

現在，ネットワークインターフェースといえば，それはEthernetを指すであろう．このもっとも普及しているEthernetとは，どのような信号が流れているのだろうか．

そこで次号では，このEthernetを物理的なレベルから理解するために，ケーブル上にどのような信号が流れているか電気的仕様やEthernetフレームについて解説する．そして10Base-Tに対応したLANコントローラを，実際にFPGAで設計してみる．またそのコントローラを制御するドライバも作成する．

## 編集後記

●毎年，年の瀬になると，1年経つのも早いものだなぁと思います．毎年買っていた年末ジャンボ宝くじも今年は頭の中が錯綜していて，すっかり忘れていました．初夢宝くじにすることにします．年末のクリスマス商戦に間に合ったPSXでお正月はゲーム三昧でしょうか．いや，やっぱり寝正月でしょうね．（檀）

●今年は何かと事件が多く，たいへんな一年になってしまったが，何とか無事に越せそうで何より．思い返してみると，確かに最悪に近い年だったのだが，幸運なこともあった．その幸運のおかげで今現在も無事でいられるようなものだ．周りの方々に深く感謝したい気分だ．本当にありがとうございました．（=IO）

●ある冬，家に入れ忘れたザリガニが凍ってしまい，動揺して落としたところ真っ二つに．割れた部分をくっつけて，そのまま外に置いてしまった．春，ザリガニを確認したところ，なんと生きていた！しかも，割れた部分がちゃんとくっ着いていたんだよ．…この話をしても誰も信じてくれないので悲しい．（もみ）

●今まで映画とはただで見るものだった．友人が映画館で働いていたので招待券を毎月戴いていた．しかし最近は枚数が減り，週末には使えなくなり，とうとう招待券そのものが廃止されてしまった．これからは自腹で映画を見るとなると本数が減るな．今まで何百本もただ見させてくれてありがとう——．（太陽熱）

●最近自宅に間違い電話がよくかかってきます．しかも同じ人宛に!でも，かけてくる人はバラバラ(若い男性，中年風，おばあちゃん?も)．ひどいのは「違います」と言ってるのに，数分後にまたかかってきたり(怒)．全員その人に連絡を取りたがっている風なんですが…．事情を聞こうとすると切られるし…何事?（M）

●出張で生まれて初めて京都へ．デザインされた街並みの美しさに感心．建物の高さ制限がなされていることや，整然としたマス目の道路，そして方向が互い違いになっている一方通行路…生まれてこの方，拡張に次ぐ拡張でグチャグチャになった街並みが当然だった身には驚きでした．やはり初期設計が肝心か．（み）

●目覚まし時計が鳴らなくなったと思っていたら，どうも無意識に止めていたようで…．やはり長年使っているとアラームの音にも慣れてしまうのでしょう．でも新しい時計にしようと思っても好みの時計が見つからないし…．しばらくはこのまま今ある目覚まし時計に頼るしかないんでしょうね(苦笑)．（Y2）

●私は物作りが好きなのだが，最近興味を持ちはじめているものにプログラミングがある．優れたフリーウェアなどを使っているとそれを作りだすためのプログラムとはどんなものか好奇心がわく．以前かじったことはあるのだがもう一度トライしてみようと思う．まずは優れた入門書を見つけなければ!（ふ）

## お知らせ

■ 読者の広場

本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

■ 投稿歓迎

本誌に投稿をご希望の方は，連絡先(自宅/勤務先)を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です(送り先はsupportinter@cqpub.co.jpまで)．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

■ 本誌掲載記事についてのご注意

本誌掲載記事には著作権があり，示されている技術には工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は，所有者の許諾が必要です．また，掲載された回路，技術，プログラムなどを利用して生じたトラブルについては，小社ならびに著作権者は責任を負いかねますので，ご了承ください．

本誌掲載記事をCQ出版(株)の承諾なしに，書籍，雑誌，Webといった媒体の形態を問わず，転載，複写することを禁じます．

■ コピーサービスのご案内

本誌バックナンバーの掲載記事については，在庫(原則として24か月分)のないものに限りコピーサービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．

- コピー料金(税込み)
  1ページにつき100円
- 発送手数料(判型に関わらず)
  1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
- 送付金額の算出方法
  総ページ数×100円＋発送手数料
- 入金方法
  現金書留か郵便小為替による郵送
- 明記事項
  雑誌名，年月号，記事タイトル，開始ページ，総ページ数
- 宛て先
  〒170-8461　東京都豊島区巣鴨1-14-2
  CQ出版株式会社　コピーサービス係
  (TEL：03-5395-4211，FAX：03-5395-1642)

■ お問い合わせ先のご案内

- 在庫，バックナンバー，年間購読送付先変更に関して
  販売部：03-5395-2141
- 広告に関して
  広告部：03-5395-2133
- 雑誌本文に関して
  編集部：03-5395-2122

記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface
©CQ出版(株)　2004　振替　00100-7-10665
2004年3月号　第30巻　第3号 通巻第321号)
2004年3月1日発行(毎月1日発行)
定価は裏表紙に表示してあります

発行人／増田久喜
編集人／山形孝雄
編集／大野典宏　村上真紀　山口光樹　大竹友美　小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／神崎真理子　森　祐子
広告／澤辺　彰　中元正夫　菅原利江

発行所／CQ出版株式会社　〒170-8461　東京都豊島区巣鴨1-14-2
電話／編集部(03)5395-2122　FAX／(03)5395-2127
広告部(03)5395-2133　URL http://www.cqpub.co.jp/interface/
販売部(03)5395-2141　E-mail supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社　美和印刷株式会社
製本／星野製本株式会社


日本ABC協会加盟誌
(新聞雑誌部数公査機構)　ISSN0387-9569